A Distributed Hash Table

by

Frank Dabek

S.B., Computer Science (2000); S.B. Literature (2000)
M.Eng., Computer Science (2001)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

September 2005

Author .....
Department of Electrical Engineering and Computer Science
November 4, 2005

Certified by .....
M. Frans Kaashoek
Professor
Thesis Supervisor

Certified by .....
Robert T. Morris
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Distributed Hash Table

Frank Dabek

Submitted to the Department of Electrical Engineering and Computer Science
on November 4, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

DHash is a new system that harnesses the storage and network resources of computers distributed across the Internet by providing a wide-area storage service, DHash. DHash frees applications from re-implementing mechanisms common to any system that stores data on a collection of machines: it maintains a mapping of objects to servers, replicates data for durability, and balances load across participating servers. Applications access data stored in DHash through a familiar hash-table interface: `put` stores data in the system under a key; `get` retrieves the data.

DHash has proven useful to a number of application builders and has been used to build a content-distribution system [34], a Usenet replacement [118], and new Internet naming architectures [133, 132]. These applications demand low-latency, high-throughput access to durable data. Meeting this demand is challenging in the wide-area environment. The geographic distribution of nodes means that latencies between nodes are likely to be high: to provide a low-latency `get` operation the system must locate a nearby copy of the data without traversing high-latency links. Also, wide-area network links are likely to be less reliable and have lower capacities than local-area network links: to provide durability efficiently the system must minimize the number of copies of data items it sends over these limited capacity links in response to node failure.

This thesis describes the design and implementation of the DHash distributed hash table and presents algorithms and techniques that address these challenges. DHash provides low-latency operations by using a synthetic network coordinate system (*Vivaldi*) to find nearby copies of data without sending messages over high-latency links. A network transport (*STP*), designed for applications that contact a large number of nodes, lets DHash provide high throughput by striping a download across many servers without causing high packet loss or exhausting local resources. *Sostenuto*, a data maintenance algorithm, lets DHash maintain data durability while minimizing the number of copies of data that the system sends over limited-capacity links.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

A Distributed Hash Table

Frank Dabek

Submitted to the Department of Electrical Engineering and Computer Science
on November 4, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

DHash is a new system that harnesses the storage and network resources of computers distributed across the Internet by providing a wide-area storage service, DHash. DHash frees applications from re-implementing mechanisms common to any system that stores data on a collection of machines: it maintains a mapping of objects to servers, replicates data for durability, and balances load across participating servers. Applications access data stored in DHash through a familiar hash-table interface: `put` stores data in the system under a key; `get` retrieves the data.

DHash has proven useful to a number of application builders and has been used to build a content-distribution system [34], a Usenet replacement [118], and new Internet naming architectures [133, 132]. These applications demand low-latency, high-throughput access to durable data. Meeting this demand is challenging in the wide-area environment. The geographic distribution of nodes means that latencies between nodes are likely to be high: to provide a low-latency `get` operation the system must locate a nearby copy of the data without traversing high-latency links. Also, wide-area network links are likely to be less reliable and have lower capacities than local-area network links: to provide durability efficiently the system must minimize the number of copies of data items it sends over these limited capacity links in response to node failure.

This thesis describes the design and implementation of the DHash distributed hash table and presents algorithms and techniques that address these challenges. DHash provides low-latency operations by using a synthetic network coordinate system (*Vivaldi*) to find nearby copies of data without sending messages over high-latency links. A network transport (*STP*), designed for applications that contact a large number of nodes, lets DHash provide high throughput by striping a download across many servers without causing high packet loss or exhausting local resources. *Sostenuto*, a data maintenance algorithm, lets DHash maintain data durability while minimizing the number of copies of data that the system sends over limited-capacity links.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

Acknowledgments

This thesis is the product of a close collaboration with my colleagues in PDOS and Project IRIS. Russ Cox was instrumental in the development of Vivaldi. David Karger provided a great deal of much-needed assistance with the theoretical aspects of Chord and DHash. Latency optimizations to DHash are the result of collaboration with Jinyang Li and Chuck Blake. Emil Sit, Frans Kaashoek, James Robertson, and Josh Cates helped build and deploy the implementation of DHash and Chord without which many of the ideas presented here would never have been developed. Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, and Byung-Gon Chun helped develop the ideas that led to Sostenuto and assisted with the presentation of Sostenuto in Chapter 6.

Some of the material presented in this thesis has been published elsewhere. The CFS filesystem is described in an SOSP publication [34]. Emil Sit authored an IPTPS paper on UsenetDHT [118]. Work on Vivaldi was done in collaboration with Russ Cox. The work began with a class project [30]; a simple version of the algorithm was presented at HotNets [31]. The height-model and adaptive timestep were first discussed in a SIGCOMM publication [33]; The analysis of DHash latency and the STP transport were first published at NSDI [35]. The material in Chapter 6 (data maintenance) is described in a paper currently under submission.

I am extremely grateful for the chance to have worked with my advisers, Frans Kaashoek and Robert Morris. I can only aspire to bring to my work the same level of clarity, rigor, and honesty that they have demonstrated during my time at MIT.

The research described in this thesis is part of the IRIS project and was funded by a grant from the National Science Foundation under NSF Cooperative Agreement No. ANI-0225660. IBM provided support in the form of a fellowship.

Frans Kaashoek Robert Morris

Dan Aguayo	John Bicket	Sanjit Biswas	Chuck Blake
Timo Burkhard	Ben Chambers	Benjie Chen	Russ Cox
Douglas De Couto	Bryan Ford	Michael Freedman	Cliff Frey
Kevin Fu	Thomer Gil	Kyle Jamieson	John Jannotti
Michael Kaminsky	Eddie Kohler	Max Krohn	Chris Laas
David Mazieres	Athicha Muthitacharoen	Eric Peterson	Sean Rhea
James Robertson	Rodrigo Rodrigues	Emil Sit	Alex Snoeren
Jeremy Stribling	Emmett Witchel	Alex Yip	Nickolai Zeldovich

Hari Balakrishnan Peter Druschel David Karger Ion Stoica

Jinyang Li

Dedicated to the memory of Joshua Cates (1977-2004).

Where are the songs of Spring? Ay, where are they?
Think not of them, thou hast thy music too…

Contents

1

Introduction

A distributed hash table (DHT) is a reliable, scalable, wide-area data storage system that frees programmers from many of the complications of building a distributed system. DHTs store blocks of data on hundreds or thousands of machines connected to the Internet, replicate the data for reliability, and quickly locate data despite running over high-latency, wide-area links. The DHT addresses problems of locating data and replicating it for reliability, which are common to many distributed systems, without additional work by the application. The DHT provides a generic interface, which makes it easy for a wide-variety of applications to adopt DHTs as a storage substrate: `put` stores data in the system under a key; `get` retrieves the data.

Distributed hash tables fill a gap in the design space of storage systems. DHTs occupy a middle ground between small systems with strong guarantees on the service they provide (such as distributed file systems) and large unorganized, best-effort systems (such as the world wide web or file sharing systems). DHTs are able to operate over a large and previously unoccupied area of the design space; the existence of a single, practical system that operates throughout this regime will make it easier to write new distributed applications.

In filling this design space gap, DHTs attempt to combine the two strands of systems research that inspired DHTs and provide the best features of both. Inspired by small, LAN-based systems (which we will call transparent distributed systems), DHTs provide probabilistic guarantees on the success of a `get` or `put` operation. At the same time DHTs can operate on the same scales as large systems designed to run on the wide-area (which we will call Internet systems).

Transparent distributed systems, such as Harp [75] and DDS [49], combine the resources of a number of machines while attempting to isolate the user from any evidence that the system is distributed. Users of a distributed file systems such as Harp would be hard-pressed to prove that their requests were being served by a cluster of machines rather than a single NFS server. To obtain this transparency these systems provide strong guarantees on the results of operations. DDS, for example, guarantees that a `get` operation will always see the results of the last `put` operation. Providing this guarantee is possible because transparent distributed systems are designed to run on a small number of machines connected to a local-area network and assume a high-bandwidth, low-latency, reliable interconnect. On the wide-area where these assumptions are violated, DDS could not make the same guarantees. DDS assumes that the network is never partitioned, for instance, a common occurrence on the wide-area. Also, techniques that these systems use to provide consistency (two-phase commit or the Paxos agreement protocol) are expensive to run over unreliable, high-latency links.

The other research ancestor of DHTs, Internet systems, share with transparent distributed systems the goal of aggregating the resources of multiple machines. However, Internet systems focus not on transparency, but on maximizing scale. These systems are designed to run on a

large number of geographically distributed nodes on the wide-area network. The mechanism of these systems is often visible to the user and only best-effort guarantees are offered. The scalability of these systems comes at the cost of lax guarantees: a file-sharing service like Gnutella offers a keyword search interface but may fail, even in normal operation, to return the location of a file even if it does exist in the system [14].

DHTs are a result of the desire for a system with the scale (large number of machines, large amount of data, large inter-node latency) of Internet systems and the graceful failure masking and strong guarantees of transparent distributed systems. DHTs sacrifice some of the guarantees of transparent distributed systems and can not scale as well as loosely organized systems like the web, but the result is a system that will run in a wide range of environments, from machine room clusters to a collection of cable modem users spread across the Internet, and provide a generic interface that is a suitable substrate for a large number of distributed storage applications that can operate with eventual consistency. The lack of strong consistency has not limited the usefulness of DHash: scenarios that necessitate strong consistency, such as write sharing, are rare and applications designed to run on DHash can be designed to minimize the amount of mutable data in the system and eliminate the possibility of multiple writers [86].

Because DHTs work in a variety of deployment scenarios, they provide the possibility of a near universal storage infrastructure. Any wide-area distributed system must cope with the challenges of node scale, network delay, and node failure. DHTs implement mechanisms (such as replication and efficient location algorithms) once, in the infrastructure, rather than as part of applications. The DHT interface may prove to be the storage equivalent to the IP abstraction for networks. This implementation convenience has spurred the early adoption of DHTs even in small systems where the ability of the system to route queries without maintaining full membership information is unnecessary. DHTs are attractive in these scenarios because they free the application programmer from the burden of implementing features necessary to cope with failure and also because they give the system the potential for dramatic expansion.

By designing a storage system to run on the wide-area network, we hope to take advantage of the abundance of network and disk resources connected to the Internet in the same way that file sharing networks such as Napster and Gnutella have. While decentralized, wide-area deployment provides the tantalizing potential of gaining inexpensive access to a great deal of storage and network resources there are challenges facing systems that take this approach.

Any wide-area storage system will have to cope with the sheer scale of thousands of participating nodes: a large number of nodes makes it hard to locate the data item requested by a `get` operation. Nodes are unreliable and may fail or leave the system at unpredictable times: data must be continually replicated as nodes fail to make the effect of a `put` operation permanent. Wide-area network delays and loss rates could make the latency of a `get` operation large or limit how fast bulk data can be returned to the application. In addition, in some deployment scenarios, the infrastructure for such a system may be dependent on volunteers to donate resources: these volunteers could be malicious or uncooperative and could deny access to stored data, delete it, or return incorrect values.

This thesis describes the DHash DHT that addresses many of the challenges of building a distributed storage system that runs on wide-area nodes and provides a generic interface, and useful guarantees, to applications. We will first describe the *base DHash* system and then demonstrate improvements to the system's performance and reliability. These improvements make DHash a practical platform for building applications: DHash has proven useful to a number of application builders [133, 118, 132, 34]. The rest of this chapter provides

Function	Description
`put_h(block)`	Computes the block's key by hashing its contents, and sends it to the key's successor server for storage.
`put_s(block, pubkey)`	Stores or updates a signed block; used for root blocks. The block must be signed with the given public key. The block's Chord key will be the hash of `pubkey`.
`get(key)`	Fetches and returns the block associated with the specified Chord key.

Table 1.1: DHash client API

an overview of DHash and describes how it can be used by applications. We outline the challenges facing DHash in more detail and sketch how the thesis addresses them.

1.1 DHash overview

This thesis will describe one DHT (DHash) in detail. Many of the algorithms and optimizations implemented by DHash can (and have been [92]) adopted in other DHTs. Here we give an overview of a basic implementation of DHash (base DHash). Later chapters present modifications to the basic design that improve performance and reliability. For example, the invariant governing data placement will be relaxed to improve maintenance costs.

1.1.1 DHash interface

Applications using DHash link against a DHash library which exports a durable hash table interface consisting of two operations: `put` and `get`. The `put` operation stores a piece of data into the hash table associated with a key; keys are drawn from a large (160-bit) key space. The `get` operation returns the block associated with the supplied key. Table 1.1 shows the API that DHash presents to applications.

The keys used by the DHash system could, in principle, be any 160-bit number, but they are usually the hash of the data item being stored. Using the content hash allows the system to certify the data received: if the content-hash of the returned data matches the key, the data must not have been tampered with. We use a 160-bit hash function so that collisions are overwhelmingly unlikely. Content-hash blocks are immutable; by nature, they cannot be altered without changing the key under which they are stored. Because the blocks are immutable, DHash does not need to implement any additional mechanism (e.g., two-phase commit) for providing a consistent view of data stored in content hash blocks.

The other key type supported by DHash is the hash of a public key. The data stored under a public key is signed by the corresponding private key; DHash verifies this signature when the data is retrieved to certify the data. Unlike content-hash blocks, the application can modify that data stored under the hash of a public key without changing the name of the block. DHash does not provide strong guarantees on the behavior of public-key blocks under simultaneous operations. For instance, while a write is in progress a simultaneous reader might see the newly-written value before the write operation returns to the writer. Simultaneous writes could result in either, or both, values being stored on the replicas initially. DHash runs a stabilization protocol that guarantees that eventually one of the values will be copied to all replicas. Because the guarantees on public-key blocks are weak, systems using mutable data in DHash usually construct single-writer data structures [86]. Public key

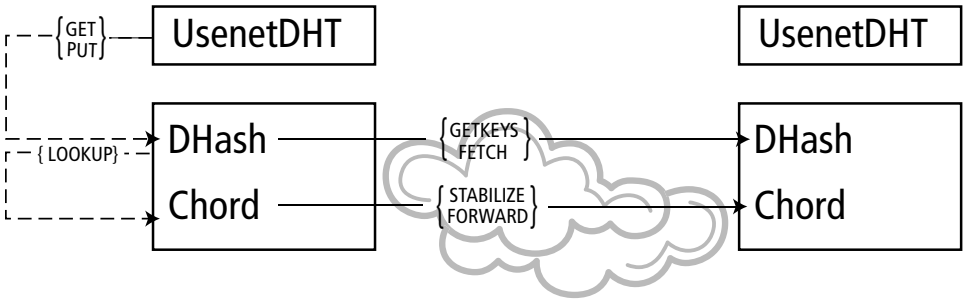


Figure 1-1: Overview of the DHash system. Applications (such as UsenetDHT) issue GET commands to DHash to find the value associated with a key; these local calls are indicated by dotted lines in the above figure. DHash, in turn, issues LOOKUP requests to Chord to locate the node responsible for the key. The Chord layer coordinates nodes to answer the lookup query by forwarding the request (solid black lines indicate remote communication via RPC); the Chord layer also exchanges periodic stabilization messages to keep routing structures up to date. The DHash layer causes nodes to exchange messages to download block data (FETCH) and to synchronize replica sets (GETKEYS).

blocks are often used to name the root of a tree-like structure of content hash blocks: this allows the system to give a single stable name to a large amount of mutable data while only using a single mutable block.

An application uses DHash by linking against a library that exports the DHash API. The library sends messages to a DHash server that coordinates with other DHash servers running on the wide-area network to resolve the application's request. The organization of the DHash system is shown in Figure 1-1.

1.1.2 Chord

DHash is built as a layer over the Chord distributed lookup system [123]. Each node participating in the Chord ring is assigned an identifier from a 160-bit circular key space; data items are assigned keys from the same space. Chord provides a scalable-lookup service that maps each 160-bit key to a node. The $lookup(k)$ function returns the ID and IP address of the node currently responsible for the key k. The node responsible for a key is the node with the identifier which most closely follows the key in the circular key space; we refer to this node as the *successor* of k and to the several nodes after k as the successor list of k. In Figure 1-2, the successor of key K19 is node N24 and the successor list of the key is nodes N24, N34, and N41. Note that N18 (the predecessor of the key) maintains pointers to all of these nodes and can definitively return K19's successor list.

Chord maintains a routing table of $\log N$ pointers to other nodes in the system and can resolve a mapping by sending $\log N$ messages, where N is the number of nodes in the system. Because Chord keeps a small amount of state, it is able to maintain the state efficiently in large or unstable systems. Section 4.1 explains how Chord implements the lookup operation using this routing table in more detail.

1.1.3 Base DHash

While Chord provides a location service, DHash is responsible for actually storing the data, creating replicas, and maintaining those replicas as disks fail. DHash has considerable leeway in deciding how to accomplish these tasks; this section examines how base DHash chooses to address these tasks along with alternative choices.

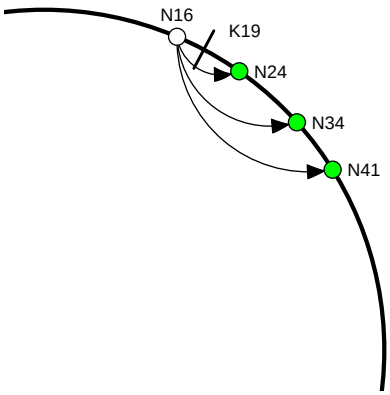


Figure 1-2: Successor relationship. The successor of a key is the first node clockwise after that key in the circular identifier space. In this example, N24 (shown as a filled circle) is the immediate successor of K19 (shown as a hash). Nodes N34 and N41 are in the successor list of key K19. Node N16 (white circle) is able to authoritatively determine the successors of any key in the range $[16, 24)$, by maintaining pointers to the nodes immediately following it on the ring. In this example and the following, an 8-bit identifier space is used.

DHash is designed to store a large number of small blocks spread approximately uniformly across participating servers. DHash uses the Chord mapping to determine which node stores a given key: the data associated with a key k is stored on the node returned by $lookup(k)$ and replicated on that node's successors. By storing replicas on the successor list, base DHash distributes data uniformly at random over all servers: a block's key is essentially random (the SHA-1 of the block's value) and node IDs are random. The result is that blocks (and load) are uniformly spread over the DHT nodes and that a block's replicas or fragments are widely scattered to avoid correlated failure. An alternative to placing replicas on the successor list would be to randomly select nodes to serve as the replica set for a key; this approach requires that the system store some meta-data that serves as a level of indirection [10].

Some DHTs provide only a key location service and let each application decide where (or even whether) to store data [60]. By using a semi-structured name space [56, 57], a system could give applications control over where a block is stored. Applications might choose to give hints about the geographic or administrative domains in which data should reside. DHash, and other DHTs that store data uniformly across all nodes in the system, are appropriate for applications that wish to view the DHT as a network storage system that automatically provides replication and location services. Our motivating examples OverCite [125] and UsenetDHT [118] are examples of this class of applications. We will describe these applications in more detail in Section 1.2.

DHTs that store data must decide on the size of the units of data to store. A DHT key could refer to a disk-sector-like block of data [34], to a complete file [112], or to an entire file system image [29]. Large values reduce number of lookups required to fetch a large amount of data. Small blocks spread the load of serving popular large files. For these reasons, and because some applications (such as SFR [132]) require the DHT to store small blocks, DHash is optimized to store blocks of 8 KB or less. The main advantage of large blocks is potentially higher throughput; one might expect that the need to perform a DHT lookup for each 8K block would lead to poor performance. However, as we will show in Chapter 5, DHash is able to take advantage of available network capacity even when using small (8K) blocks.

Storing data on the successor list ensures good load-balance but could limit the throughput that an application obtains when it downloads a large file that has been broken down into

blocks. Because base DHash rigidly maps a given block to some node, the slowest node in the system will be the owner of some blocks and therefore must be periodically contacted. The slowest node dictates how fast the entire download can progress on average: N times the throughput of the slowest node if there are N nodes in the system. Note that if the slowest node is a bottleneck, this implies that the access link of the node performing the download is N times faster than the access link of the slowest node.

The system might achieve a higher initial rate by downloading from higher-capacity hosts at those hosts' full rate; however, such a strategy does not change when the last block will be received from the slowest host. We also might relax the requirement on where blocks are stored to avoid allowing slow hosts to limit the system performance. This quickly leads to a question of access control: should all hosts be allowed to participate fully in the system? If the system membership is open, but the system favors high-performance hosts, it is not clear what benefit the overall system derives from allowing the slower hosts to participate (the penalty for allowing such hosts is clear, however). One possibility is that hosts with slow access links can be used as "emergency" replicas: normally they are not contacted during a download, but if the better-equipped replicas are not available, the resource constrained hosts can service downloads slowly to maintain block availability.

Data stored in the DHT must be replicated for reliability. Any storage system should provide two distinct but related properties: *durability* and *availability*. The definition of availability is straightforward: a block is available at some point in time if an application can access the block through the system at that time. The definition of durability is a bit more difficult: we'll define a block to be durable if it is stored on some media and will be available through the system at some point in the future. DHash's main goal is data durability; however, since the system, in practice, can only observe availability failures it also maintains data availability.

The number of copies of each block that are maintained helps to determine the probability that a data block will be lost; increasing the replication factor reduces the probability of loss but increases the amount of data that must be sent over the network when a block is initially stored and during the ongoing process of replacing failed replicas. DHash is designed to operate on a large number of nodes that suffer frequent, temporary failures. Making multiple copies of a block when it is inserted into DHash makes data available despite a fixed number of node failures, but if data is to be preserved during a process of continuous membership churn the system must maintain the number of replicas over time by creating new copies of data when existing copies are lost. DHash's data movement algorithms maintain desired replication levels by making new copies of data when nodes fail. DHash's replication algorithm minimizes the number of copies needed to maintain durability; notably, the algorithm tries to avoid creating replicas in response to temporary failures that do not affect durability. Base DHash also creates new copies of data items when nodes join or leave the system to maintain the invariant that blocks are stored on the successor list of a key; one optimization to base DHash will be relaxing this invariant to avoid moving data every time a node joins or leaves the system.

Base DHash is implemented as follows. Each block is stored as 14 erasure-coded fragments, any seven of which are sufficient to reconstruct the block, using the IDA coding algorithm [104]. The 14 fragments are stored at the 14 immediate successors of the block's key. When an application calls `get(key)`, the originating node performs a Chord lookup, which ends when the key's predecessor node returns the key's 16 successors; the originating node then sends seven parallel requests the first seven successors asking them each to return one fragment. Note that, in the example of Figure 1-2 data is replicated only three times:

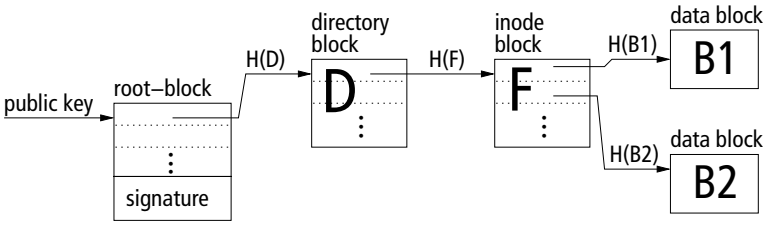


Figure 1-3: A simple CFS file system structure example. The root-block is identified by a public key and signed by the corresponding private key. The other blocks are identified by cryptographic hashes of their contents.

in the implementation of DHash the data associated with K19 would be stored at 14 nodes following 19 in identifier space.

1.2 Using DHash

A number of applications have been designed based on DHash. We present an overview of some of them here to illustrate how DHash is used in practice.

1.2.1 A cooperative file system

The cooperative file system (CFS) is a peer-to-peer read-only storage system that provides provable guarantees for the efficiency, robustness, and load-balance of file storage and retrieval. Publishers insert a file system into CFS under a 160-bit key; clients access the file system read-only by specifying the key. CFS would be useful to open-source developers who wish to distribute popular software and source code, for example. It also illustrates some abstractions common to applications which use DHash.

CFS stores a file system in DHash in much the same way that a traditional file system is stored on a disk with 160-bit content-hash identifiers replacing block numbers to identify blocks. Blocks stored as part of a file system are either file system data blocks or file system meta-data, such as directories. Because all pointers in the file system are content hashes, the hashes of blocks higher in the tree can be used to authenticate all data beneath that block. The root block, which can authenticate the entire file system, is named by the publisher's public key instead of by its content hash; the publisher then signs the root block with his private key to authenticate it. The identifier associated with the root block can be thought of as the name of the file system. If we named the root block by content hash, the file system's name would change every time it was updated; we use a public key to name the root block so that the file system has a consistent name. Figure 1-3 shows how CFS stores a file-system in DHash. CFS's use of a tree of hashes is based on earlier work: the SFSRO file system [44] and Merkle trees [83].

CFS has weaker semantics than a traditional filesystem because DHash does not make strong guarantees about the data returned by a read of a mutable, replicated block (such as the public key block used to name the root of the CFS file system). DHash only guarantees that, barring network partitions, one value will eventually be associated with each key. It is possible that a user reading CFS while an update is taking place, or shortly after, will see stale (but internally consistent) data.

CFS assumes that DHash does not aggressively garbage collect data. In fact, a DHash node never deletes data unless it has no remaining disk space; at that point the least-recently-

used item is deleted. In practice, we don't have experience with running DHash when disk space is at a premium and, given the rate at which disk capacity grows, we don't expect that disk space will be a scarce resource. If disk space was scarce, techniques have been developed to fairly allocate it among users of the DHT [109].

We expect that most applications which store bulk data in DHash will create similar tree-like structures out of small (8K) blocks: a tree of content-hash blocks with a public-key block at the root is a common abstraction for DHash-based applications. Storing bulk data as small blocks spreads load evenly and makes parallel fetch possible.

1.2.2 UsenetDHT

Usenet is a distributed bulletin board service: users can post and read articles on a variety of topics. Usenet was one of the first distributed systems and remains popular: about 1.4 TB of data are posted to Usenet each day. Today, most of that data is in the form of binary files (pornography and illegally distributed movies), but the system still hosts many popular message boards.

Usenet replicates each article at each news server, requiring each server to download that 1.4 TB of data per day (or at about 100Mbps, 24 hours a day). UsenetDHT is a system that reduces the storage and bandwidth resources required to run a Usenet server by spreading the burden of data storage across participants. UsenetDHT stores data in a DHash system running on the news servers. By using DHash, UsenetDHT dramatically reduces the storage and bandwidth required to run a news server: the amount of data that must be stored on each node participating in UsenetDHT scales inversely with the number of participating nodes since the nodes share the cost of storing the data. Each node's bandwidth requirements are proportional to the fraction of articles read by clients of that news server rather than to the total number posted by all users of the system. While an index of articles could be stored in DHash, the system preserves the existing flood-fill mechanism of Usenet to distribute an index where article IDs are replaced with content-hash identifiers. Preserving the existing flood-fill mechanism for meta-data lets news server administrators use existing filtering techniques (as long as filters depend only on the article header) and minimizes the amount of mutable data stored in DHash.

1.2.3 OverCite

OverCite [125] is a distributed replacement for the CiteSeer digital research library [47]. OverCite distributes the storage and network requirements of CiteSeer among a collection of participating organizations (the current CiteSeer is maintained at a single American university). OverCite uses DHash to store the full-text of research papers as well as organize meta-data about document citations. The creators of OverCite believe that it can be a model for deploying web sites that are too large or popular to be hosted by a single, resource-constrained, organization. A DHT provides a way to organize the resources of a number of organizations into a single entity.

1.2.4 Rendezvous applications

CFS and UsenetDHT store a large amount of data in DHash. Other applications, such as the semantic-free reference system [132], take advantage the flat identifier space provided by DHash. SFR uses DHash identifiers as a persistent name for web-pages that might move

between servers. SFR stores public-key blocks that contain the current location of the named object. Other applications such as i3 [121] and DOA [133] use a DHT to store small values used for rendezvous.

1.3 Problem statement

The base DHash system and the CFS file system described above were implemented as earlier work [32]. CFS provided good load balance and was scalable in the number of hops required to perform a lookup operation. However, deployment of the system revealed that it was not usable for our target applications. Experience with the system identified a number of challenges: 1) low latency lookup, 2) high throughput when fetching bulk data in parallel, and 3) low maintenance overhead in the face of membership change.

The remainder of this thesis will describe algorithms and optimizations to DHash that improve its performance, reliability, and efficiency. Although we have described a layered architecture, these optimizations will involve modifying Chord as well as DHash. We'll explain these problems and the corresponding solutions in the context of DHash, but any system that stores data on the wide-area must address the same concerns. This section explains, in detail, problems with the initial implementation; Section 1.4 will outline solutions to the problems discussed here.

1.3.1 Using Vivaldi to reduce latency

Base DHash suffered from high lookup latency because Chord treats all hops as equal even though the distribution of RTTs between the nodes in the test bed is wide. Ideally, the latency of a DHash lookup would be the same as the latency of an IP RTT from the requester to the node storing the requested data item. We will define the ratio of the latency of a DHash lookup to the IP RTT to be the *stretch* of the lookup. Chord and DHash, as described, do not make decisions about where to route queries or data based on the location of the querier or the destination. As a result Chord lookups may traverse distant nodes even if the querier and target are nearby. Since each hop Chord makes is a random sample of the latencies on the test bed and Chord makes $\log N$ hops per lookup, the base Chord system has a stretch of about $\log N$ while our goal is a stretch of 1 (or even less).

1.3.2 Improving throughput with STP

Some of the applications using DHash store a large amount of data that will be downloaded in bulk (large movie files, for instance). DHash stripes this data across many machines, and traditional stream-oriented network protocols proved not to be suitable for downloading data. DHash provided low throughput and caused high loss on the network.

The initial DHash implementation used TCP [61] as a transport. As the number of nodes in the system grew, this strategy quickly led to resource exhaustion: because a large file is striped over many machines DHash was forced to contact, and make a TCP connection to, a large number of machines in a short period of time. The system quickly exhausted the available operating system resources. This resource exhaustion was one of the first scalability barriers that DHash faced.

Even if these resource shortages are avoided (by aborting TCP connections, for instance) using TCP has a number of drawbacks. For example, TCP has a high connection setup cost. Also, keeping a large number of short-lived TCP connections open led to unfair bottleneck

link sharing and high packet loss due to congestion. When TCP is used to transfer only a very small amount of data the connections remain in slow-start and the congestion avoidance properties of the transport are not effective. Using UDP as a transport eliminates the first problem but is equally prone to congest links leading to high loss rates and low throughput. Maintaining TCP connections to just the hosts in a node's routing table avoids resource consumption problems but requires sending data over multiple overlay links (including in and out of intermediate nodes' access links) instead of directly to the requesting node; this leads to increased latency and decreased throughput.

A good transport for the DHT must keep enough data in flight to cover the network's delay-bandwidth product, stripe data over multiple slow access links in parallel, recover in a timely fashion from packet loss, and allow direct communication with any node in the system. The transport must also provide congestion control in order to avoid overflowing queues and causing packet loss. These goals are similar to those of traditional unicast transport protocols such as TCP, but with the additional requirement that the solution function well when the data is spread over a large set of servers.

Building such a transport requires addressing a set of problems not faced by unicast protocols. There is no steady "ACK clock" to pace new data, since each message has a different destination. The best congestion window size is hard to define because there may be no single delay and thus no single bandwidth-delay product. Quick recovery from lost packets via fast retransmit [120] may not be possible because replies are not likely to arrive in order. Finally, averaging round-trip times to generate time-out intervals may not work well because each message has a different destination.

1.3.3 Maintenance

Storing data reliably by replicating it induces a network traffic cost: as nodes fail new copies of data must be made to replace those lost if replication levels are to be maintained; data may also need to be moved when nodes join since the mapping of keys to nodes implemented by Chord changes. Because data must be recreated by sending replicas over the network when disks fail, the disk failure rate puts a hard limit on the amount of data that can be stored in DHash. Only as much data as can be copied in a node lifetime can be stored per-node in the best case.

Base DHash creates a fixed number of replicas on the successors of a key. It uses a number of replicas larger than the expected number of simultaneous failures to ensure that at least one copy of the data survives at catastrophic event and aggressively moves data to the current set of successors whenever that set changes (due to either a node joining or leaving the system). Base DHash deletes additional replicas beyond the minimum requested. This strategy causes base DHash to unnecessarily transfer data in the common case in which a node fails temporarily but returns with disk contents intact. In this case, the system repeatedly made an extra copy of the data when it could have simply waited for the existing copy to return. Likewise, when a node joined temporarily the system moved data to the node needlessly. Since the goal of DHash is to provide data durability, the ideal maintenance system would only create new copies of data in response to permanent failure (disk failure, for example). By making unnecessary copies, base DHash limits the amount of data it could store for a given node and disk failure rate.

1.4 Solutions

These problems limit the usefulness of DHash (or any wide-area storage system). This thesis presents algorithms that solve these problems and allow DHTs to run in a wide range of environments by reducing `get` latency despite the high inter-node latencies and maximizing the amount of data that the system can store given limited capacity links and unreliable nodes. One theme of these solutions will be adaptation: DHash works in much of the feasible design space because it has few fixed parameters and attempts, as much as possible, to adapt its configuration to the environment in which it is running.

1.4.1 Vivaldi

In addressing these challenges, we found that many optimizations needed accurate latency predictions. A DHash node will often be able to choose to complete an operation by communicating with any node from a possible set of nodes; the node performing the operation would prefer to send the message to the lowest delay node. For example, a node would benefit from knowing the latency to a potential node to decide if it is the closest replica of a data item or a good choice for addition to a proximity routing table. In a large, dynamic system such as DHash, it is likely that the node has not communicated with any of the potential nodes in the past and thus does not have a prior measurement of the delay to any of the nodes.

Vivaldi is a synthetic coordinate system in the style of systems like GNP ([90, 97, 115, 28]) that allows a node to make a prediction about the round-trip time to another node without a direct measurement to the node in question. Vivaldi is fully decentralized (most other systems require specially designated landmark nodes), adapts to changing network conditions, and finds coordinates that predict RTT well. Vivaldi's predictions are accurate enough to be useful: the median relative error of Vivaldi's predictions is about 12 percent when predicting Internet RTTs.

Hosts participating in Vivaldi compute synthetic coordinates in some coordinate space such that distance between two hosts' synthetic coordinates predicts the RTT between them in the Internet. Thus, if a host x learns the coordinates of hosts y and z and wishes to know which is closer in the network, x doesn't have to perform explicit measurements to determine the RTT to y and z. Instead, the delay from x to y and z can be predicted by distances in coordinate space and the node with the lower predicted delay can be chosen.

The ability to predict RTT without prior communication allows systems to use proximity information for better performance with less measurement overhead (in terms of both network traffic and time required for measurements) than sending measurement probes to determine latency directly. DHash uses Vivaldi coordinates to select which of a number of replicated servers to fetch a data item from; in this example, it would not be practical to first probe all the servers to find the closest, since the delay required to wait for the probes' return would be greater than the reduction in the latency of the operation that would from a choice informed by the probes' information. DHash also uses coordinates to set a timeout that determines when a message should be retransmitted.

While unrelated to the design of DHTS, the structure of Vivaldi coordinates also gives us some insights on the structure of the Internet. For example, in our data set there are few routes than run through central Asia; as a result the Internet can be modeled well by a flat coordinate space as opposed to a spherical one. Vivaldi also incorporates a new coordinate space that models the effect of slow access links; this new coordinate space is more accurate than Euclidean spaces with the same number of degrees of freedom.

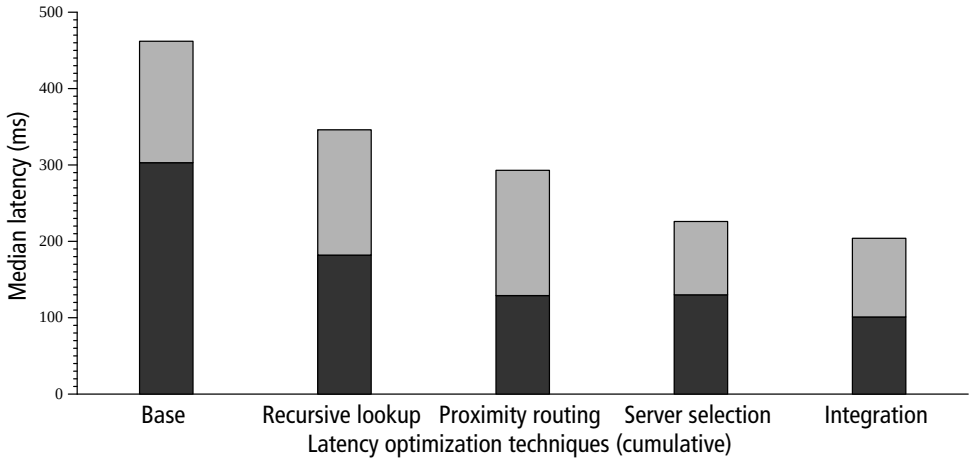


Figure 1-4: The cumulative effect of successive optimizations on the latency of a DHash data fetch. Each bar shows the median time of 1,000 fetches of a randomly chosen 8192-byte data block from a randomly chosen host. The dark portion of each bar shows the lookup time, and the light portion shows the time taken to fetch the data. These data are from the implementation running on PlanetLab.

1.4.2 Latency

DHash can reduce the stretch of a lookup by taking network proximity into account when deciding which nodes to place in its routing table. This optimization, *proximity neighbor selection*, has been discussed in earlier work [50]. To this literature, we add an exploration of why PNS achieves the gains it does.

The main result of our analysis of proximity routing is that the latency of a routing hop is related to the size of the set of possible next-hop nodes; by maximizing choice at each stage of the lookup, lookup latency can be minimized. We also observe that because later hops in a lookup become exponentially more restricted, the latency of a lookup is dominated by the last few hops. This observation implies that there is a limit to how well any proximity scheme can do given the distribution of latencies on the Internet. DHash will always suffer a stretch of around 1.5. Similarly, it means that the size of the network does not dramatically affect the latency of lookups in an overlay. One might expect lookup latency to grow with the log of network size, but, in fact, it remains relatively constant.

Based on these observations we present modifications to Chord (which will also apply to other multi-hop lookup systems) that sample possible next-hop nodes to produce routing tables that contain nearby nodes and avoid making hops that have few choices.

To achieve a low stretch the system must also locate a nearby copy of the data. DHash selects the replica with the lowest delay when choosing between multiple replicas of a block. DHash also combines the operations of routing to data and retrieving it to further reduce stretch. A lookup can "stop early" when enough fragments have been discovered or "overshoot" a distant predecessor to collect more fragments from nearby successors to avoid high delay hops. These techniques allow the system to avoid high-latency hops to the predecessor of a key.

Figure 1-4 gives a preview of the results of adding proximity awareness improvements to DHash. Each pair of bars shows the median time to fetch a block on the PlanetLab [25] test-bed after cumulatively applying each design improvement. The design improvements shown are recursive rather than iterative routing, proximity neighbor selection, fetching of

data from the closest copy, and integration of lookup routing and data fetching. Recursive routing improves latency by eliminating unnecessary return trips from intermediate nodes on the lookup path. Proximity neighbor selection chooses Chord fingers that are nearby in the network; as a result the latency of each hop is reduced. Server selection reduces latency by choosing a replica of the block that is close to the requester based on Vivaldi coordinates. Finally, by integrating Chord and DHash more tightly we can further reduce *get* latency by avoiding an expensive last hop to the key's predecessor. These design improvements together reduce the total fetch latency by more than a factor of two.

1.4.3 Throughput

DHash addresses the transport problem by using a new protocol, the Striped Transport Protocol (STP). STP allows nodes to achieve high throughput for bulk data transfers while downloading a blocks from a large number of nodes. STP allows data to be downloaded directly from other nodes and works well in a very large system because it does not maintain any per-destination state; instead, all of its decisions are based on aggregate measurements of recent network behavior, and on Vivaldi latency predictions. STP's core mechanism is a per-node (rather than per-connection) TCP-like congestion window controlling the number of concurrent outstanding RPCs. Treating a striped download as a single flow allows the system to use a single UDP socket instead of many TCP connections. It also causes a striped download to compete fairly with TCP streams for bottleneck link capacity and causes low packet loss.

STP works best when the bottleneck link is the downloading node's access link (we expect that this will be common). In this case, the single window STP maintains corresponds to the access link's capacity. If the actual bottleneck is inside the network, STP may fail to fully utilize the access link but will not cause excessive packet loss. We evaluated STP on the RON [5] test-bed. Both the median throughput and the range of individual node throughputs are higher when using STP compared to a transport that maintains persistent TCP connections to nodes in the routing table.

1.4.4 Sostenuto: Replica maintenance

DHash reduces the bandwidth required to maintain replication levels by lazily creating and tracking extra replicas beyond a base number of eagerly-maintained replicas. The presence of extra replicas lets DHash avoid the sending new copies of data over the network when replicas temporarily fail. Sostenuto is a maintenance algorithm that adaptively chooses the number of extra replicas such that the system creates as few new replicas as possible due to temporary failures.

Sostenuto is parameterized by the number of replicas that the system attempts to maintain at all times. To understand how large this number must be to preserve block durability we have developed a model based on a constant process of block failure and repair. This birth-death model captures the basic fact about durability: to safely store data it must be repaired faster than it is destroyed by disk failures. Earlier work [32] considered the response of the system to a single, massive simultaneous failure and predicted a much higher replication level than is necessary.

To avoid the cost associated with new nodes entering the system, Sostenuto relaxes the invariants governing where data must be placed. We also present changes to the Merkle synchronization protocol [19] to maintain its efficiency under this relaxed placement scheme.

1.5 Contributions

The main contribution of this thesis is an efficient wide-area storage system that facilitates the construction of distributed applications. DHash provides a familiar hash-table interface that makes programming applications easy. Because DHash runs in a wide variety of environments, application deployments based on DHash can graduate from a small cluster of machines to an Internet-wide deployment as they grow in popularity.

Several new algorithms were developed as part of this effort: Vivaldi, STP, and Sostenuto solve problems likely to be faced by any wide-area storage system. All of these algorithms are implemented as part of the DHash implementation that is available for download by the public [24]. As a result of these optimizations, DHash is able to support our target applications: UsenetDHT is currently supporting a partial news feed running on 100 PlanetLab machines. The `get` latency of the system, the time that a user must wait to read a selected article, is approximately 200ms. By implementing Sostenuto, we reduced traffic caused by data maintenance by a factor of 6. OverCite has been deployed on a cluster of a dozen machines spread between MIT and the RON testbed.

The Vivaldi network coordinate system differs from all previous network coordinates systems by being fully decentralized. Because it is decentralized, Vivaldi is easily integrated into distributed systems, as demonstrated by is adoption by other systems [93, 92].

Our analysis of latency optimizations in Chord contributes an understanding of the lookup latency that $O(\log N)$-hop protocols can expect when using PNS. We show that the median lookup latency using PNS is approximately $3 \times \delta$ (where δ is the median latency in the system) and is independent of the size of the system.

STP is an algorithm for congestion control suitable for systems that download large amounts of data from many servers. A number of lightweight transports have been developed to avoid TCP resource consumption problems, but all are stream oriented (STCP [91], for example). STP is the first transport protocol designed explicitly for systems that perform many-to-one striped downloads.

The Sostenuto replication system addresses the problem of replication in large-scale, wide-area systems where nodes fail frequently but often return with intact disks. Prior work in replication focused on systems that comprise a small number of well-connected, reliable nodes; these designs perform poorly in the wide-area. Sostenuto is also likely to use fewer wide-area network resources than earlier wide-area replication systems because it contains fewer parameters that must be configured by the operator. These parameters depend on the properties of the environment which can not be known in advance; configuring them incorrectly leads to inefficient operation.

An additional contribution of this work is a large-scale measurement study of Internet latencies. The result of this study is a data set consisting of the latencies between 1740 DNS servers on the Internet. This data set has used by other researchers to explore DHT design [72, 73].

1.6 Rest of thesis

The rest of this thesis describes, in detail, how we improved base DHash. Chapter 2 outlines the our experimental environment. We introduce Vivaldi in Chapter 3 and show how it can be used to reduce the latency of `get` operations in Chapter 4. Chapter 5 describes how STP lets DHash obtain high-throughput by downloading from many sources in parallel.

The subject of Chapter 6 is data maintenance: we outline a model that lets system designers choose the right replication level and show an algorithm, Sostenuto, that maintains durability while minimizing the number of additional replicas sent over the network. Chapter 7 surveys related work and Chapter 8 concludes.

2

System environment

Distributed storage systems have been designed for deployment across a wide range of configurations. For example, early file-sharing systems targeted a large number of weakly connected nodes (possibly dial-up users). Distributed disk systems like Frangipani [128] and Petal [70] are designed to run on a few servers connected by a high-speed local interconnect; Grid storage systems run on a small number of machines connected by a fast wide-area network [22].

Although DHash works well in a wide-variety of environments, this thesis we will focus on deploying DHash on a wide-area cluster of well-connected (> 1Mbit/s link) servers. We expect that these machines will occasionally suffer network disconnection and reboots. More rarely, disks fail. While the system handles unexpected node departure, we expect that the DHash software will be run as a service, rather than an application. That is, we assume that as long as a machine is running, the DHash daemon is running on that machine. This configuration will allow DHash to store a large amount of data reliably.

2.1 Evaluation setup

We realized this deployment scenario by implementing DHash and deploying it on the PlanetLab testbed. PlanetLab [25] is a global research test bed comprising approximately 600 machines distributed at almost 300 sites around the world. Machines are currently located in North America, South America, Europe, Asia, and Australia. A significant fraction of these machines are located at American universities with high-speed Internet2 links. However, each application is limited to using 1.5Mbps per node by a traffic shaper.

DHash has been running on PlanetLab off and on since 2001. The experiments in this thesis were performed over several years, during which time the testbed expanded and changed its resource allocation policy (in particular, the 1.5Mbps cap was instituted in February, 2005). This section will characterize the testbed as it existed in the summer of 2005.

Much of the experimental evaluation of the system (detailed in later chapters) was performed by measuring the implementation running on PlanetLab. Measuring the running implementation provides a high degree of realism: real, lossy links are being used and nodes frequently fail due to hardware faults, operator error, and software bugs.

Our implementation experience has been shaped by the characteristics of PlanetLab. PlanetLab has undeniable strengths: it is large (by the standards of what has been traditionally available to the academic community) and well-distributed across the Internet. In the sense that it runs a standard operating system (Linux) and standard hardware (x86) and sends packets across the same network links shared by millions of users, PlanetLab is "real."

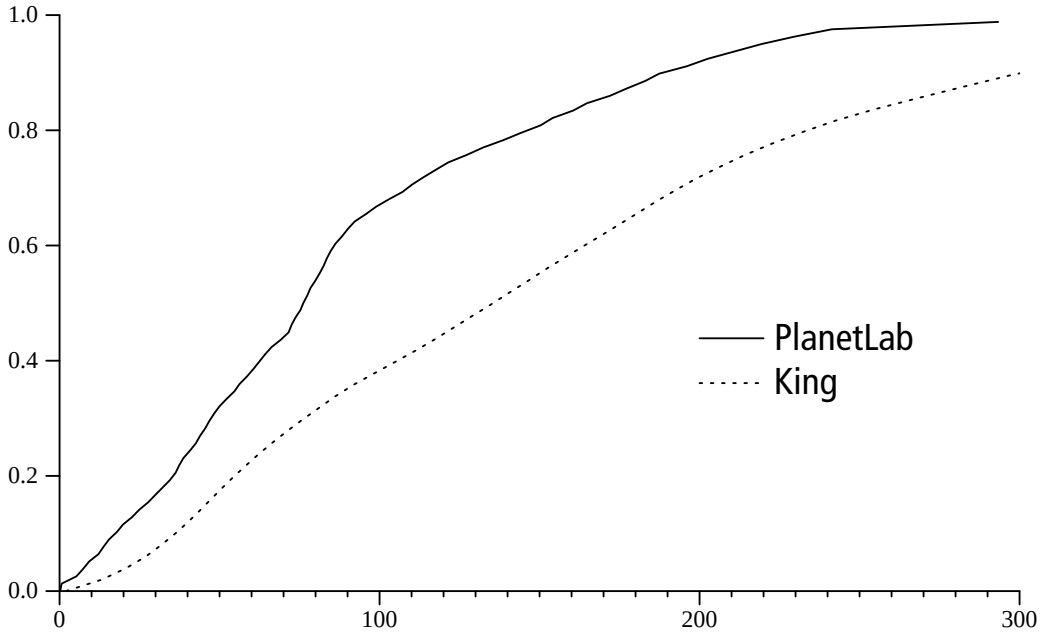


Figure 2-1: Cumulative distribution of pair-wise latencies for all pairs of nodes in the King and PlanetLab data sets.

However, PlanetLab's size is something of a mismatch for the goals of DHash. In testing DHash on PlanetLab we made a conscious decision to design a system for thousands of nodes while testing on a test bed made up of only a few hundred nodes. STP, for example, is designed under the assumption that the number of nodes in the system is, essentially, infinite. When run on 300 nodes, but with an assumption that the number of nodes is infinite, STP limits the state it maintains unnecessarily. Designed to run on 300 nodes, STP would certainly keep state about all nodes in the system. By keeping more state, STP could set better, per-host retransmission timers and maintain per-host congestion windows allowing it to take advantage of high-capacity peers.

PlanetLab nodes are more heavily used (load averages of more than 30 are not uncommon) than then nodes we expect DHash to be run on in practice. High load makes it difficult to make precise and repeatable latency measurements and hard to obtain high throughput. For some experiments (notably the throughput comparison in Chapter 5, we used the RON testbed. The RON testbed is smaller, consisting of less than 30 machines, but has a much smaller user population and is correspondingly less loaded.

We also evaluated portions of the system in a packet level simulator, p2psim [46]. This simulator lets us evaluate the performance of latency optimizations in a stable setting and quickly implement potential optimizations. Vivaldi is also evaluated using p2psim. While p2psim is suitable for understanding latency optimizations, it does not model link capacity or queuing delay and would not be suitable for understanding the effects of, for instance, throughput optimizations.

2.2 Latency

The latency between nodes will affect DHash's ability to quickly locate a data item. Because DHash uses Chord to locate an object, and Chord sends multiple ($\log N$) messages per lookup request, in a network where the inter-node latency is high, we might expect high latency

lookup operations. In this section we characterize the internode latencies on PlanetLab and in the latency matrix that drives our simulations.

2.2.1 PlanetLab

The PlanetLab nodes span the globe, but most are located at North American universities with fast Internet2 connections. The median RTT of the PlanetLab data set is 76ms; this low RTT represents the bias towards nodes located in the United States.

Figure 2-1 shows the cumulative distribution of the all pair-wise latencies between nodes on PlanetLab: this graph has $O(N^2)$ points for N PlanetLab nodes. This data is taken from a publically available measurement service [124]. Some of the the analysis of optimizations to Chord and DHash depend on the shape of this distribution. Notably, the analysis of latency optimizations described in Section 4 depends on the fact that latencies are roughly uniformly distributed. This distribution arises because latency on the Internet is dominated by geographic distance and nodes are spread randomly across the face of the Earth. The distance between randomly chosen points on the surface of a disc is not distributed uniformly, but it is a close enough approximation of uniform for our purposes. A uniform distribution might not arise in, for example, a corporate network consisting of large two offices connected by a high latency, long haul line.

2.2.2 King: latencies for p2psim

The biggest challenge we face in obtaining realistic results from p2psim is determining internode latencies. One approach to setting the latencies between nodes is to use an Internet topology model such as ITM [13], BRITE [82], etc. These models are unsatisfactory because they only specify a connectivity among nodes; the user must choose edge weights and a routing policy. It is unclear how to choose the parameters to effectively model Internet latencies. When evaluating Vivaldi we are especially averse to creating a model (Vivaldi coordinates) of a model (ITM) of the Internet.

Instead of using a model, we choose to measure the round-trip-times between 1740 DNS servers on the Internet directly; we use DNS servers because we are able to force two unmodified servers to report the latency between the servers. This approach provides us with a great degree of realism, but is limited in scale. Another limitation of this approach is that it measures the latency between DNS servers rather than random nodes. It is not clear if these servers are representative of the average node on the Internet. In particular, one might expect that these servers are better connected than an average node. The servers are, however, well-distributed geographically.

We built a tool based on the King method [51] to collect the full matrix of RTTs. To determine the distance between DNS server A and server B, we first measure the round trip time to server A and then ask server A to recursively resolve a domain served by B. The difference in times between the two operations yields an estimate of the round trip time between A and B (see Figure 2-2). Each query involves a unique target name to suppress DNS caching. Previous work [51] confirms the accuracy of this method; we additionally spot-checked our tool by using it to measure latencies that we could also measure directly (e.g., when one nameserver was on MIT's network).

We harvested the addresses of recursive DNS servers by extracting the NS records for IP addresses of hosts participating in a Gnutella network. By collecting addresses from a large, distributed system we hope to obtain a collection of name servers that is representative of the

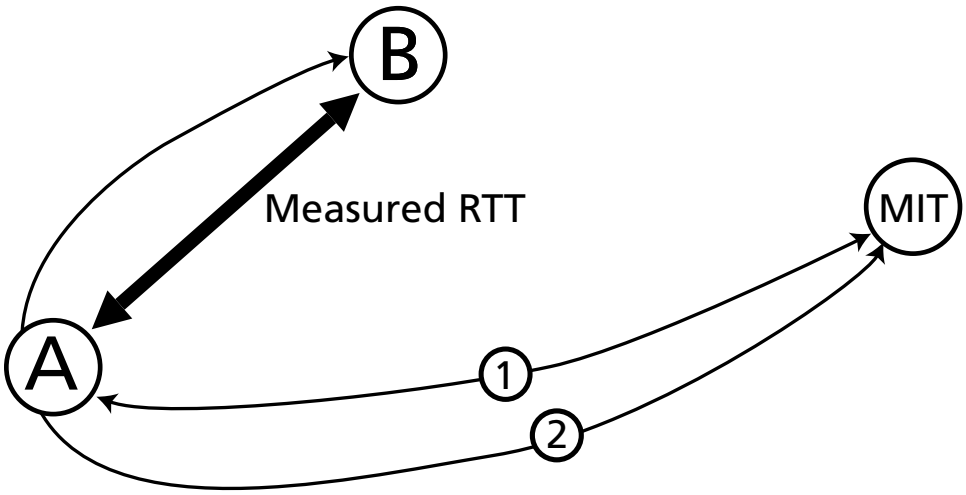


Figure 2-2: It is possible to measure the distance between two nameservers by timing two DNS queries. The first query (1) is for a name in the domain of nameserver A. This returns the latency to the first nameserver. The second query is for a name in the domain nameserver B (2) but is sent initially to the recursive nameserver A. The difference between the latency of (1) and (2) is the latency between nameserver A and B.

distribution of Internet hosts. Because we crawled the Gnutella network from the vantage point of the United States, it is possible that our sample biased towards North American hosts. If a domain is served by multiple, geographically diverse name servers, queries targeted at domain D (and intended for name server B) could be forwarded to a different name server, C, which also serves D. To avoid this error, we filtered the list of target domains and name servers to include only those domains where all authoritative name servers were on the same subnet (i.e. the IP addresses of the name servers were identical except for the low octet). We also verified that the target nameservers were responsible for the associated names by performing a non-recursive query for that name and checking for the "aa" bit in the response header, which indicates an authoritative answer.

We measured pairwise RTTs continuously, at random intervals, over the course of a week. Around 100 million measurements were made in total. We compute the final RTT for a given pair as the median of all trials. Using the median RTT filters out the effects of transient congestion and packet loss. Other measurement studies [90] have used the minimum measured RTT to eliminate congestion effects; this approach is inappropriate for the King method since congestion can cause measured RTT to be higher or lower than the true value. The King method can report a RTT lower than the true value if there is congestion on the path to the first nameserver. The median RTT of the King data set is 159ms.

Some nameservers were obvious outliers in the data set: the latency to these servers was equal and small from all hosts. This inaccuracy could be the result of high load on the nameservers themselves or heavy queuing near the servers. If load or queuing at name server A adds a delay that is significantly larger than the network latency, the initial query (to A) and recursive query (via A to B) will require roughly the same amount of time and the estimated latency between that server and any other server will be near zero. We identified these servers by the disproportionate number of triangle inequality violations they participated in and removed them from the data set. About 10 percent of the original nodes were removed in this way.

The King data set provides a larger set of nodes and greater geographic diversity. Figure 2-

Dates	12 Aug 2004 – 6 Jan 2005
Number of hosts	409
Number of failures (reboots)	13356
Number of data-loss failures	645
Average host downtime (s)	1337078
Failure inter-arrival time (s)	143, 1486, 488600
Crash inter-arrival time (s) (Median/Mean/Max)	555, 16670, 488600

Table 2.1: CoMon+PLC trace characteristics

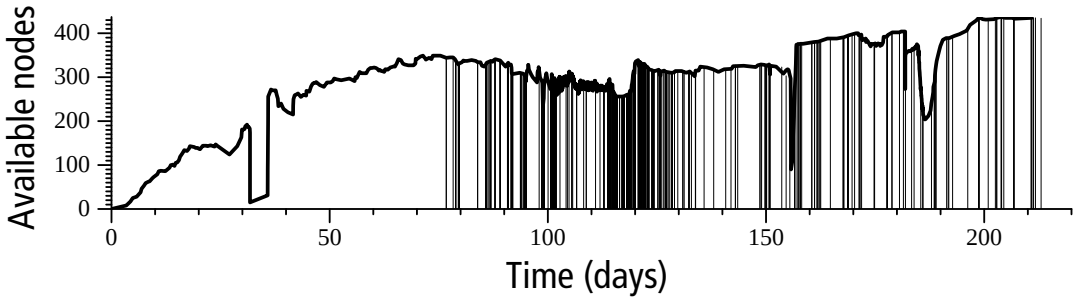


Figure 2-3: PlanetLab hosts available over time

1 shows the cumulative distribution of both data sets. Because PlanetLab is biased towards American universities, its median latency is significantly lower than that of the King data set.

2.3 Node reliability

Understanding the reliability of PlanetLab nodes is also important; node reliability impacts Chord's ability to maintain routing information and DHash's ability to produce new data replicas to replace lost replicas. We are interested in both the number of temporary failures and the number of permanent (disk) failures. For instance, if nodes frequently fail permanently, the system will frequently be required to make a large number of new data replicas to replace lost data: if the failure rate is too high the bandwidth required to create the replicas may exceed the link capacity and the system may not be able to "stay ahead" of failures.

To understand the failure characteristics of our chosen test-bed, we use a detailed trace from historical data collected by the CoMon project [95] on PlanetLab [25]. CoMon monitors all PlanetLab hosts every five minutes, allowing failures to be detected quickly. Further, CoMon reports the actual uptime counter from each machine, allowing the time at which each machine reboots can be determined precisely.

In order to identify disk failures (resulting typically from operator actions such as operating system reinstallations or disk upgrades), the CoMon measurements were supplemented with event logs from PlanetLab Central [98]. Disk failure information was available only after 25 October 2005. Table 2.1 summarizes the statistics of this trace.

Figure 2-3 plots the number of PlanetLab nodes that were available during our 25 week trace. A vertical line indicates that data was lost during the failure; most of these data loss events are due to upgrades rather than hardware failure.

Even though PlanetLab nodes are maintained by host institutions, the trace includes a large number of node and disk failures. Many of the disk failures are due to periodic upgrades

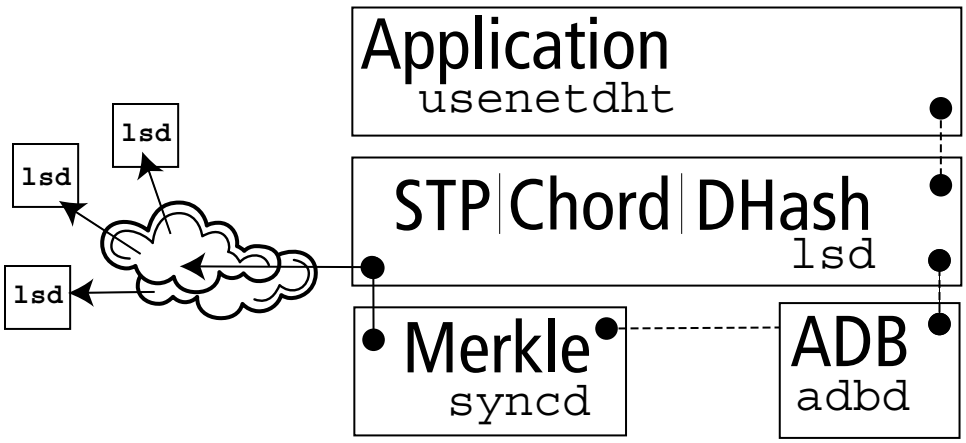


Figure 2-4: The DHash implementation. Boxes represent UNIX processes and are labeled with the process name (lower right) and functionality implemented by the process (center). Processes communicate via local RPC over UNIX domain sockets (dotted lines) and via the network (solid line). The `lsd` process implements the Chord lookup primitive as well as DHash. Both Chord and DHash send RPCs using the STP transport protocol (discussed in Chapter 5). `adbd` provides an asynchronous database service; data is stored in a DB4 ("sleepycat") database. A separate process (`syncd`) monitors and maintains replication levels of DHash objects (see Chapter 6);

of the nodes' operating systems. Correlated failures, a circumstance for which our algorithm is not designed, are more frequent than we expect. These correlated failures arise mainly because the test bed is centrally administered. Operating system upgrades, for instance, cause many nodes to reboot at once or disks to be reformatted. Bugs in the deployed kernel or security breaches have caused many or all nodes to be unavailable for an extended period of time. This trace ends prior to a particularly large upgrade in which 200 machines were failed in a short time period.

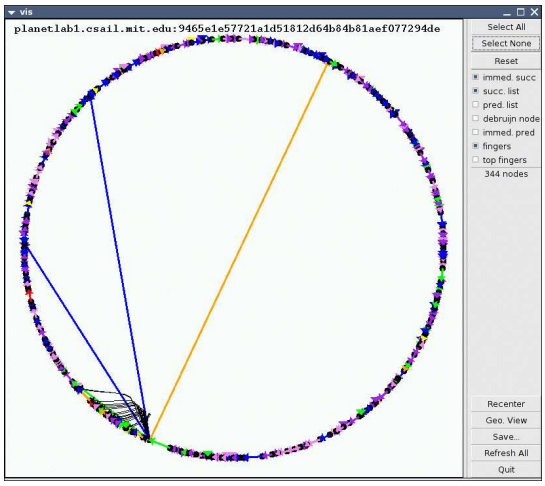
2.4 Implementation

DHash is written in C++ in an asynchronous style using the libasync library [81]. The core of the system (DHash, Chord, and supporting libraries) comprises slightly less than 25,000 lines of code. DHash and Chord run as a daemon (`lsd`) and communicate with local applications via local RPC (a client library provides an asynchronous function call interface to applications).

Figure 2-4 shows the organization of the DHash implementation. The `lsd` process implements the logic of Chord and DHash and handles remote communication with other nodes using STP's congestion control mechanisms. Two helper processes accompany `lsd` and communicate with the `lsd` via local RPCs. One process (`syncd`) monitors the replication level of blocks stored by the local node using the Merkle synchronization protocol; the other (`adbd`) provides a non-blocking interface to the DB4 (sleepycat) database that stores blocks.

Short-comings of PlanetLab (mainly due to scale and load) forced a number of implementation decisions that may have to be reconsidered if DHash is used in the environment for which it is ostensibly designed (a large number of volunteer hosts). Because PlanetLab has become the *de facto* test bed for evaluating distributed systems, it is important to understand how PlanetLab differs from the intended deployment and what parts of the DHash implementation constitute PlanetLab-specific engineering.

For instance, the extremely high load on PlanetLab hosts leads to very large ($>$ 1 second)



(a)

```
 === successors
0.  ee17f75ed9 planetlab1.csail.mit.edu
1.  ee2e42285a i30clyde.ibds.uni-karlsruhe.de
2.  eed36a535c planetlab3.cs.uoregon.edu
3.  efa4c4a15a 130.192.201.30
4.  efea7d0703 planetlab15.Millennium.Berkeley.edu
5.  f08937d3dc planetlab2.cs.Virginia.edu
6.  f1b31408c5 planetlab2.cs.Virginia.edu
7.  f26014d9d4 planetlab1.georgetown.edu
8.  f2889038fa csplanetlab3.kaist.ac.kr
=== fingers
0.  ee17f75ed9 planetlab1.csail.mit.edu
1.  ee2e42285a i30clyde.ibds.uni-karlsruhe.de
2.  efea7d0703 planetlab15.Millennium.Berkeley.edu
3.  f1b31408c5 planetlab2.cs.Virginia.edu
4.  f28b568dd9 planetlab2.comet.columbia.edu
5.  f7ad4ce6f6 lefthand.eecs.harvard.edu
6.  ffadb8152d 192.208.48.5
7.  132183695e planetlab2.cs.umass.edu
8.  3863ad70ed planetlabtwo.ccs.neu.edu
9.  757c406046 planetlab3.comet.columbia.edu
```

(b)

Figure 2-5: (a) The Chord visualizer (`vis`). The fingers and successor list of a node running on an MIT PlanetLab node are shown. (b) The output of `nodeq` showing the routing table for the same node in text form.

latencies when writing to disk. This forced us to move all disk operations to a helper process (`adbd`) that communicates with the primary daemon via local RPCs. On a less loaded system, this complexity would likely be unnecessary. It could be argued that this complexity is the result of our decision to use an asynchronous programming style coupled with the poor support by UNIX for non-blocking I/O on disks. Many, including this author, have argued that it is easier to construct systems in an async style, when compared to a threaded implementation, because async systems are largely devoid of data races. In the case of DHash, the async approach was a mixed blessing. While it is hard to say what would have been more difficult to accomplish with threads, hiding the high latency of disk access was certainly a drawback.

Building and maintaining a large distributed system proved challenging. One appeal of DHTs is that they insulate application developers from these challenges by implementing mechanisms common to all distributed storage systems.

The process of turning our original prototype into a continuously-running service was greatly aided by a series of utilities that allowed developers to examine the state of the running ring or of individual nodes. Figure 2-5 shows the output of two tools. `vis` displays a graphical representation of a running Chord ring; `nodeq` displays information about a single node's routing table.

3

Vivaldi

Several of the optimizations described in this thesis depend on the Vivaldi synthetic coordinate system. Synthetic coordinate systems [90, 97, 115, 28] allow an Internet host to predict the round-trip latencies to other hosts without directly measuring the latency. Hosts compute synthetic coordinates in some coordinate space such that the distance between two hosts' synthetic coordinates predicts the RTT between them in the Internet.

The key property of coordinate systems is the ability to quickly (i.e., in much less than the RTT) obtain an estimate of the RTT. The ability to predict RTT without prior communication allows systems to use proximity information for better performance with less measurement overhead than probing. Operations that complete in about an RTT (such as fetching a small data item) could be infeasible to optimize without coordinates: the time required to make a measurement or to query an on-line system like IDMaps [42] or Meridian [137] would outweigh the benefits of selecting a nearby replica.

Coordinates can benefit a variety of systems, however: a coordinate system can be used to select which of a number of replicated servers to fetch a data item from; coordinates are particularly helpful when the number of potential servers is large or the amount of data is small. In either case it would not be practical to first probe all the servers to find the closest, since the time spent performing the probes would be more than the time saved by an intelligent choice. Content distribution and file-sharing systems such as KaZaA [65], BitTorrent [27], and CoDeeN [134] are examples of systems that offer a large number of replica servers. CFS [34] and DNS [84] are examples of systems that offer modest numbers of replicas, but operate on small data items. All of these applications could benefit from network coordinates.

The Internet's properties determine whether synthetic coordinates are likely to work well. For example, if Internet latency is dominated by speed-of-light delay over links, and the Internet is well-enough connected that there is a roughly direct physical path between every pair of hosts, and the Internet routing system finds these direct paths, then synthetic coordinates that mimic latitude and longitude are likely to predict latency well.

Unfortunately, these properties are only approximate. Packets often deviate from great-circle routes because few site pairs are directly connected, because ISPs peer at a limited number of locations, and because transmission time and router electronics delay packets. The resulting distorted latencies make it impossible to choose two-dimensional host coordinates that predict latency perfectly, so a synthetic coordinate system must have a strategy for choosing coordinates that minimize prediction errors. Coordinates need not be limited to two dimensions; Vivaldi is able to eliminate certain errors by using higher dimensional coordinates or augmenting coordinates with a *height*.

Designing a synthetic coordinate system for use in large-scale distributed Internet applications involves the following challenges:

- Finding a metric space that embeds the Internet with little error. A suitable space must cope with the difficulties introduced by Internet routing, transmission time, and queuing.

- Scaling to a large number of hosts. Synthetic coordinate systems are of most value in large-scale applications; if only a few hosts are involved, direct measurement of RTT is practical.

- Decentralizing the implementation. Many emerging applications, such as peer-to-peer applications, are distributed and symmetric in nature and do not inherently have special, reliable hosts that are candidates for landmarks.

- Minimizing probe traffic. An ideal synthetic coordinate system would not introduce any additional network traffic, but would be able to gather whatever information it needed from the application's existing communication.

- Adapting to changing network conditions. The relative location of a host in the synthetic coordinate space may change due to congestion or even reconfiguration of the network. The system should be able to adjust the coordinates of hosts periodically to respond to these changes.

A number of previous synthetic coordinate systems address some of these challenges, but none addresses them all.

A further contribution of this chapter is a new model for synthetic coordinates: coordinates drawn from a two-dimensional Euclidean model with a height can accurately predict latency between the 1740 Internet hosts. Simulations show that this model is better than 2- or 3-dimensional Euclidean models or a spherical model. These findings suggest that the following properties hold in the data set: inter-host RTT is dominated by geographic distance, the Internet core does not "wrap around" the Earth to any significant extent, and the time required to traverse an access-link is often a significant fraction of total RTT.

3.1 Vivaldi algorithm

Vivaldi assigns each host synthetic coordinates in a coordinate space, attempting to assign coordinates such that the distance in the coordinate space between two hosts accurately predicts the packet transmission RTT between the hosts. No low-dimensional coordinate space would allow Vivaldi to predict RTTs between Internet hosts *exactly*, because, for example, Internet latencies violate the triangle inequality. The algorithm instead attempts to find coordinates that minimize the error of predictions.

We first describe this prediction error in more detail and briefly discuss possible coordinate systems. Then, we show a simple centralized algorithm that finds coordinates that minimize a squared error function given complete knowledge of RTTs in the network. Then we present a simple distributed algorithm that computes coordinates based on measurements from each node to a few other nodes and refine this distributed algorithm to converge quickly to accurate coordinates. Finally, we explore how the choice of coordinate space affects the accuracy of Vivaldi's predictions and propose a new coordinate space that eliminates some errors.

3.1.1 Prediction error

Let L_{ij} be the actual RTT between nodes i and j, and x_i be the coordinates assigned to node i. We can characterize the errors in the coordinates using a squared-error function:

$$E = \sum_i \sum_j \left(L_{ij} - \left\|x_i - x_j\right\|\right)^2 \tag{3.1}$$

where $\left\|x_i - x_j\right\|$ is the distance between the coordinates of nodes i and j in the coordinate space. This is the standard squared error objective function. Other systems choose to minimize a different quantity; PIC [28], for instance, minimizes squared relative error. The choice of error function does not have a large impact on the accuracy of the embedding. We chose the squared error function because it has an analogue to the displacement in a physical mass-spring system: minimizing the energy in a spring network is equivalent to minimizing the squared-error function. Our algorithm for computing coordinates will be based on a rough simulation of a mass-spring system.

3.1.2 Synthetic coordinate structure

Algorithms can choose the structure of coordinates and the distance function that determines the predicted latency given two coordinates. Coordinates should be compact and it should be easy to compute an RTT prediction given two coordinates. The simplest choice is to use n-dimensional coordinates with the standard Euclidean distance function. Spherical, toroidal, hyperbolic and other coordinate structures have also been proposed (e.g., [116]). These coordinate systems use alternative distance functions in the hope that they predict latency better. Section 3.4 will introduce the height-vector coordinates. In the remainder of this section, however, we will present algorithms that work with any coordinate system that supports the magnitude, addition, and subtraction operations.

3.1.3 Centralized algorithm

We first describe a simple, centralized algorithm than can minimize Equation 3.1 given the latencies of all links in the system. Vivaldi is a distributed version of this algorithm that computes coordinates after each node measures only a subset of the links. Given our choice of an error function, E, simulating of a network of physical springs produces coordinates that minimize E.

Conceptually, this minimization places a spring between each pair of nodes (i,j) with a rest length set to the known RTT (L_{ij}). The current length of the spring is considered to be the distance between the nodes in the coordinate space. The potential energy of such a spring is proportional to the square of the displacement from its rest length: the sum of these energies over all springs is the squared error function.

Since the squared-error function is equivalent to spring energy, we can minimize it by simulating the movements of nodes under the spring forces. It will not be necessary to simulate the movement of the masses extremely faithfully (we will ignore the impact of momentum for instance): only the final configuration is of interest. While the minimum energy configuration of the spring system corresponds to the minimum error coordinate assignment, it is not guaranteed that the simulation will find this global minimum.

This approach to minimization mirrors work on model reconstruction [59] and a similar recent coordinate approach using force fields [115]. Graph layout techniques also use mass-spring models to layout nodes in a graph ([8] provides a summary of these techniques).

We will now describe the centralized algorithm more precisely. Define F_{ij} to be the force vector that the spring between nodes i and j exerts on node i. From Hooke's law [58] we can show that F is:

$$F_{ij} = \left(L_{ij} - \left\|x_i - x_j\right\|\right) \times \mathrm{u}(x_i - x_j).$$

The scalar quantity $\left(L_{ij} - \left\|x_i - x_j\right\|\right)$ is the displacement of the spring from rest. This quantity gives the magnitude of the force exerted by the spring on i and j (we will ignore the spring constant). The unit vector $\mathrm{u}(x_i - x_j)$ gives the direction of the force on i. Scaling this vector by the force magnitude calculated above gives the force vector that the spring exerts on node i.

The net force on i (F_i) is the sum of the forces from other nodes:

$$F_i = \sum_{j \neq i} F_{ij}.$$

To simulate the spring network's evolution the algorithm considers small intervals of time. At each interval, the algorithm moves each node (x_i) a small distance in the coordinate space in the direction of F_i and then recomputes all the forces (because the velocity of the masses is not saved between iterations, the simulation is not faithful to physical reality). The coordinates at the end of a time interval are:

$$x_i = x_i + F_i \times t,$$

where t is the length of the time interval. The size of t determines how far a node moves at each time interval. Finding an appropriate t is important in the design of Vivaldi.

Figure 3-1 presents the pseudocode for the centralized algorithm. For each node i in the system, compute_coordinates computes the force on each spring connected to i (line 1) and adds that force to the total force on i (line 2). After all of the forces have been added together, i moves a small distance in the direction of the force (line 3). This process is repeated until the system converges to coordinates that predict error well.

This centralized algorithm (and the algorithms that will build on it) finds coordinates that minimize squared error because the force function we chose (Hooke's law) defines a force that is proportional to displacement. If we chose a different force function, a different error function would be minimized. For instance, if spring force were a constant regardless of displacement, this algorithm would minimize the sum of (unsquared) errors.

3.1.4 The simple Vivaldi algorithm

The centralized algorithm described in Section 3.1.3 computes coordinates for all nodes given all RTTs. If the RTTs are already known, latency prediction isn't necessary; ere we extend the algorithm so that each node computes and continuously adjusts its coordinates based only on measured RTTs from the node to a handful of other nodes and the current coordinates of those nodes.

Each node participating in Vivaldi simulates its own movement in the spring system. Each node maintains its own current coordinates, starting with coordinates at the origin (at the

```
// Input: latency matrix and initial coordinates
// Output: more accurate coordinates in x
compute_coordinates(L, x)
  while (error (L, x) > tolerance)
    foreach i
      F = 0
      foreach j
        // Compute error/force of this spring. (1)
        e = L_ij − ‖x_i − x_j‖
        // Add the force vector of this spring to the total force. (2)
        F = F + e × u(x_i − x_j)
      // Move a small step in the direction of the force. (3)
      x_i = x_i + t × F
```

Figure 3-1: The centralized algorithm.

start of the simulation all nodes are located at the origin so the force vectors are not defined; when two nodes are located at the same position the system chooses a random force vector). Whenever a node communicates with another node, it measures the RTT to that node and also learns that node's current coordinates.

The input to the distributed Vivaldi algorithm is a sequence of such samples. In response to a sample, a node allows itself to be pushed for a short timestep by the corresponding spring; each of these movements reduce the node's error with respect to one other node in the system. As nodes continually communicate with other nodes, they converge to coordinates that predict RTT well.

When node i with coordinates x_i learns about node j with coordinates x_j and measured RTT rtt, it updates its coordinates using the update rule:

$$x_i = x_i + \delta \times \left(rtt - \|x_i - x_j\|\right) \times \mathrm{u}(x_i - x_j).$$

This rule is identical to the individual forces calculated in the inner loop of the centralized algorithm. The timestep in the centralized algorithm corresponds to the δ in this algorithm.

Because all nodes start at the same location, Vivaldi must separate them somehow. Vivaldi does this by defining $\mathrm{u}(0)$ to be a unit-length vector in a randomly chosen direction. Two nodes occupying the same location will have a spring pushing them away from each other in some arbitrary direction.

Figure 3-2 shows the pseudocode for this distributed algorithm. A node invokes the simple_vivaldi procedure whenever a new RTT measurement is available. simple_vivaldi is passed an RTT measurement to the remote node and the remote node's coordinates. The procedure first calculates the error in its current prediction to the target node (line 1). The node will move towards or away from the target node based on the magnitude of this error; lines 2 and 3 find the direction (the force vector created by the algorithm's imagined spring) the node should move. Finally, the node moves a fraction of the distance to the target node in line 4, using a constant timestep (δ).

This algorithm effectively implements a weighted moving average that is biased toward more recent samples; since all nodes in the system are constantly updating their coordinates, old samples eventually become outdated.

```
// Node i has measured node j to be rtt ms away,
// and node j says it has coordinates x_j.
simple_vivaldi(rtt, x_j)
  // Compute error of this sample. (1)
  e = rtt − ||x_i − x_j||
  // Find the direction of the force the error is causing. (2)
  dir = u(x_i − x_j)
  // The force vector is proportional to the error (3)
  f = dir × e
  // Move a a small step in the direction of the force. (4)
  x_i = x_i + δ × dir
```

Figure 3-2: The simple Vivaldi algorithm, with a constant timestep δ.

3.1.5 An adaptive timestep

The main difficulty in implementing Vivaldi is ensuring that it converges to coordinates that predict RTT well. The rate of convergence is governed by the δ timestep: large δ values cause Vivaldi to adjust coordinates in large steps. However, if all Vivaldi nodes use large δ values, the result is typically oscillation and failure to converge to useful coordinates. Intuitively, a large δ causes nodes to jump back and forth across low energy valleys that a smaller delta would explore.

An additional challenge is handling nodes that have a high error in their coordinates. If a node n communicates with some node that has coordinates that predict RTTs badly, any update that n makes based on those coordinates is likely to increase prediction error rather than decrease it.

We would like to obtain fast convergence and avoid oscillation. Vivaldi does this by varying δ depending on how certain the node is about its coordinates. To maintain an estimate of how accurate its coordinates are, each node maintains an estimate of the current average error of predictions made by its coordinates (the "estimated error"). Each node compares each new measured RTT sample with the RTT predicted by the current coordinates, and maintains a moving average of the recent relative errors (absolute error divided by actual latency) of these predictions. In our experiments, the estimate is always within a small constant factor of the actual error.

When a node is still learning its rough place in the network (as happens, for example, when the node first joins), larger values of δ will help it move quickly to an approximately correct position. Once there, smaller values of δ will help it refine its position. The estimated error varies just as we would like δ to: when a node joins its estimated error is initialized to a large value; as it finds coordinates that predict latency its estimated error becomes smaller. A simple adaptive δ might be a constant fraction of the node's error estimate:

$$\delta = c_c \times \text{estimated error}$$

δ can be viewed as the fraction of the way the node is allowed to move toward the perfect position for the current sample. If a node predicts its error to be within $\pm 5\%$, then it won't move more than 5% toward a corrected position. On the other hand, if its error is large (say, $\pm 100\%$), then it will eagerly move all the way to the corrected position.

A problem with setting δ to the prediction error is that it doesn't take into account the

accuracy of the remote node's coordinates. If the remote node has an accuracy of $\pm 50\%$, then it should be given less credence than a remote node with an accuracy of $\pm 5\%$. Vivaldi implements this timestep:

$$\delta = c_c \times \frac{\text{local estimated error}}{\text{local estimated error} + \text{remote estimated error}}$$

Using this δ, an accurate node sampling an inaccurate node will not move much, an inaccurate node sampling an accurate node will move a lot, and two nodes of similar accuracy will split the difference.

When updating the error estimate we similarly weight the new error sample using the local node's error estimate and sampled node's error estimate. Let e_{local} be the node's current estimated error, e_{remote} be the remote node's estimated error, and e_{sample} be the relative error of the most recent prediction. The following equation computes the weighted sample error which will be combined with the current error estimate in an EWMA:

$$e_{weighted} = e_{sample}\frac{e_{local}}{e_{remote} + e_{local}} + e_{local}\frac{e_{remote}}{e_{remote} + e_{local}}$$

The effect of this equation is similar to the effect of calculating δ using a weighted average. If the sampled node has a large error estimate, the local node's error estimate will not change greatly (the coefficient on e_{local} is large). This reflects the local node's belief that, when its error estimate is low, any prediction errors are likely due to the incorrect placement of the sampled node rather than the local node.

Computing the timestep in this way provides the properties we desire: quick convergence, low oscillation, and resilience against high-error nodes.

3.1.6 The Vivaldi algorithm

Figure 3-3 shows pseudocode for Vivaldi. The vivaldi procedure runs on each node and computes the weight of a sample based on local and remote error estimates (line 1). The algorithm must also track the local error estimate. It does this using a weighted average (lines 2 and 3); this calculation is equivalent to the error estimate calculation shown above. The weighted sample is then combined with the current estimate using an exponentially-weighted moving average. The EWMA serves to smooth the estimate. The remainder of the Vivaldi algorithm is identical to the simple version.

Vivaldi is fully distributed: an identical vivaldi procedure runs on every node. It is also efficient: each sample provides information that allows a node to update its coordinates. Because Vivaldi is constantly updating coordinates, it is adaptive; if the underlying topology changes, nodes naturally update their coordinates accordingly. Finally, it handles high-error nodes. The next sections evaluate how well Vivaldi achieves these properties experimentally and investigate what coordinate space best fits the Internet.

3.2 Experimental setup

The experiments are conducted using p2psim running with RTT data collected using the King method as described in Section 2.2.2. This section presents the details of the framework used for the experiments.

// Incorporate new information: node j has been
// measured to be rtt ms away, has coordinates x_j,
// and an error estimate of e_j.
//
// Our own coordinates and error estimate are x_i and e_i.
//
// The constants c_e and c_c are tuning parameters.
vivaldi(rtt, x_j, e_j)
// Sample weight balances local and remote error. (1)
$w = e_i/(e_i + e_j)$

// Compute relative error of this sample. (2)
$e_s = \left|\|x_i - x_j\| - rtt\right|/rtt$

// Update weighted moving average of local error. (3)
$e_i = e_s \times c_e \times w + e_i \times (1 - c_e) \times (1 - w)$

// Update local coordinates. (4)
$\delta = c_c \times w$
$x_i = x_i + \delta \times \left(rtt - \|x_i - x_j\|\right) \times \mathrm{u}(x_i - x_j)$

Figure 3-3: The Vivaldi algorithm, with an adaptive timestep.

3.2.1 Latency data

To evaluate we require the matrix of N^2 latency measurements between all pairs of N nodes. The matrix of inter-node latencies are provided by the King and PlanetLab datasets; Chapter 2 describes the data sets. Vivaldi computes coordinates using a subset of the RTTs, and the full matrix is used to evaluate the quality of predictions made by those coordinates between pairs whose true RTT was not measured to compute the coordinates. These data sets are derived from measurements of real networks; they help us understand how "embeddable" real networks are in low-dimensional spaces.

We also used two synthetic data sets. The grid data set is constructed to provide a perfect two-dimensional fit; this data set is created by placing nodes in a plane; distances between the nodes are chosen to match the Euclidean distances in the plane. When fitting this data set, Vivaldi recovers the coordinates up to rotation and translation. This data set is not intended to model a realistic network topology; we'll use it to illustrate Vivaldi's behavior.

We also use the ITM topology generation tool [13] to generate topologies. The latency between two nodes in this data set is found by finding the shortest path through the weighted graph that ITM generates. This data set allows us to explore how topology changes affect Vivaldi.

Each simulated node runs an instance of Vivaldi which sends RPCs to other nodes, measures the RTTs, and uses those RTTs to run the decentralized Vivaldi algorithm.

We define the error of a link as the absolute difference between the predicted RTT for the link (using the coordinates for the two nodes at the ends of the link) and the actual RTT. We define the error of a node as the median of the link errors for links involving that node. We define the error of the system as the median of the node errors for all nodes in the system.

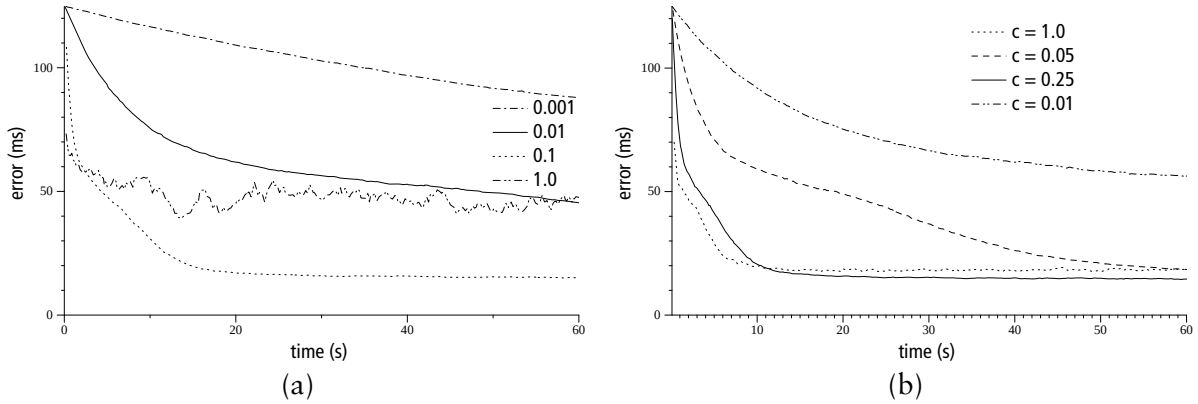


Figure 3-4: The effect of δ on rate of convergence. In (a), δ is set to one of a range of constants. In (b), δ is calculated using an adaptive δ based on the node's estimated error as described in Section 3.1.5, with c_c values ranging from 0.01 to 1.0. The adaptive δ causes errors to decrease faster.

Note that the error function we use to evaluate Vivaldi is not the error function the algorithm minimizes (squared error); we evaluate relative error mainly to ease the comparison of Vivaldi with algorithms that minimize relative error directly.

The main limitation of p2psim is that the RTTs do not vary over time: the simulator does not model queuing delay or changes in routing. Doing this typically requires modeling the underlying structure of the network. Since this research involves evaluating models for the structure of the network, it seems safest to stick to real, if unchanging, data rather than model a model.

In all of the experiments using p2psim, nodes obtain an RTT measurement by timing an RPC to the target node. Nodes keep one RPC outstanding at all times; each time a measurement is completed a new RPC is sent. In practice DHash might make measurements faster than this: DHash often send a number of RPCs in parallel.

3.3 Evaluation

This section examines (1) the effectiveness of the adaptive time-step δ; (2) how well Vivaldi handles high-error nodes; (3) Vivaldi's sensitivity to communication patterns, in order to characterize the types of network applications that can use Vivaldi without additional probe traffic; (4) Vivaldi's responsiveness to network changes; and (5) Vivaldi's accuracy compared to that of GNP. The experiments presented in this section use Euclidean coordinates; Section 3.4 investigates other coordinate systems.

3.3.1 Time-Step choice

The goal of the adaptive timestep is to find a avoid oscillation while still converging quickly. To evaluate the adaptive timestep we simulated Vivaldi on the King data set using 3-dimensional Euclidean coordinates.

Figure 3-4(a) plots the progress of the simulation using various constant values of δ. The plot shows the median prediction error as a function of time. Small values of δ, such as 0.001, cause slow convergence; increasing δ to 0.01 causes faster convergence; but increasing δ again to 1.0 prevents Vivaldi from finding low-error coordinates. The reason for the high

average error is that the high δ causes the coordinates to oscillate in large steps around the best values.

In Figure 3-4(b) we repeat the experiment using an adaptive δ based on a node's error estimate as computed in Section 3.1.5. The data show the effectiveness of using a large δ when a node's error is high (to converge quickly) and a small δ when a node's error is low (to minimize the node's oscillation around good coordinates). Empirically, a c_c value of 0.25 yields both quick error reduction and low oscillation.

3.3.2 Robustness against high-error nodes

Large numbers of newly-joined nodes with inconsistent coordinates could disrupt the coordinates of well-established nodes. Vivaldi's adaptive δ should address this problem: when a node joins, it knows its relative error is quite large, and so when it communicates with other nodes, those other nodes will approach it with appropriate skepticism.

Figure 3-5 shows the results of a simulation to test this hypothesis. The simulation uses the two-dimensional grid data set to make it easy to visualize the evolution of the system. The simulation started with 200 nodes that already knew coordinates that predicted latency well. Then we added 200 new nodes to the system and let the system evolve, using $\delta = 0.05$ in one case an adaptive time step as described in Section 3.1.5 with $c_c = 0.25$ in the other. Figure 3-5 shows the evolution of the two systems as well as the error over time. After a few iterations using the constant δ metric, the initial structure of the system has been destroyed, a result of wise old nodes placing too much faith in young high-error nodes. Because the initial structure is destroyed, existing nodes can no longer use the current coordinates of other existing nodes to predict latency until the system re-converges.

In contrast, the adaptive δ preserves the established order, helping the new nodes find their places faster. Also, because the structure of the original nodes is preserved while new nodes join, those nodes can continue to use current coordinates to make accurate predictions to other original nodes. Finally, the convergence time of the new nodes is significantly faster; they converge at $t = 60$ using the relative time-step versus $t \approx 250$ using the constant δ.

3.3.3 Communication patterns

As presented, Vivaldi relies on samples obtained from traffic generated by the application using it. To understand the range of systems in which this approach is appropriate, we must characterize the sampling necessary for accurate computation of coordinates.

Some kinds of sampling work badly. For example, Priyantha *et al.* [102] show that sampling only nearby (low-latency) nodes can lead to coordinates that preserve local relationships but are far from correct at a global scale. Figure 3-6 shows the coordinates chosen for nodes laid out in a grid when each node communicates only with its four neighbors. This case is clearly a pathological one, but we would like to characterize the boundary between normal behavior and pathology.

The pathological case can be fixed by adding long-distance communications, giving the nodes a more global sense of their place in the network. But how much long-distance communication is necessary in order to keep the coordinates from distorting? To answer this question, we ran an experiment with a grid of 400 nodes. Each node was assigned eight neighbors: the four immediately adjacent to it and four chosen at random (on average, the random neighbors will be far away). At each step, each node decides to communicate either with an adjacent neighbor or a faraway neighbor. Specifically, each node chooses, with

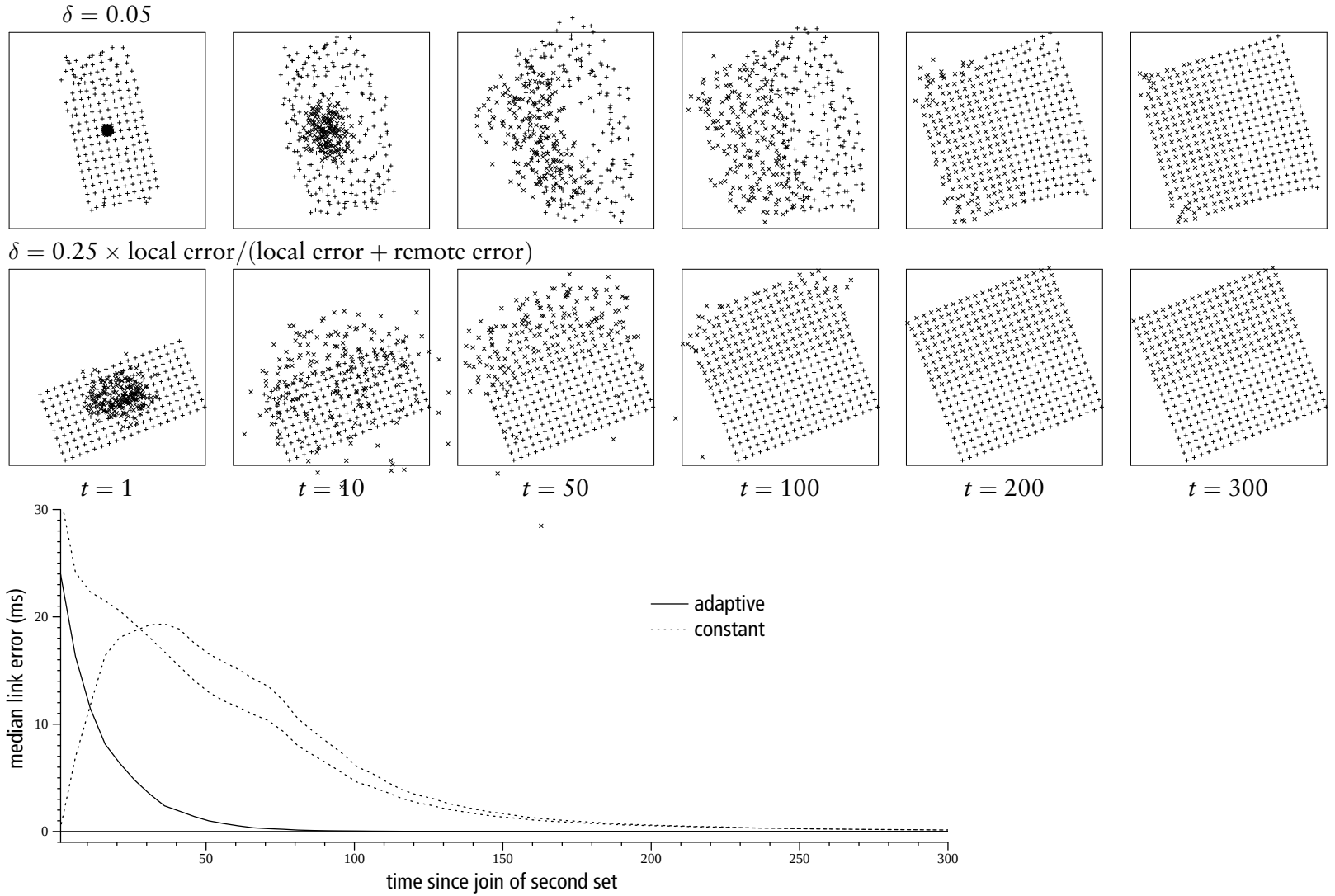


Figure 3-5: The evolution of a stable 200-node network after 200 new nodes join. When using a constant δ, the new nodes confuse the old nodes, which scatter until the system as a whole has re-converged. In contrast, the adaptive δ allows new nodes to find their places quickly without disturbing the established order. The graph plots link errors for constant (dotted) and adaptive (solid) δ. At $t = 1$, the lower line in each pair is the median error among the initial nodes. The higher line in each pair is the median error among all pairs. The constant δ system converges more slowly than the adaptive system, disrupting the old nodes significantly in the process. The median error of the of the existing nodes is not disturbed during the join when using an adaptive timestep: it remains near zero as we expect for the grid data set.

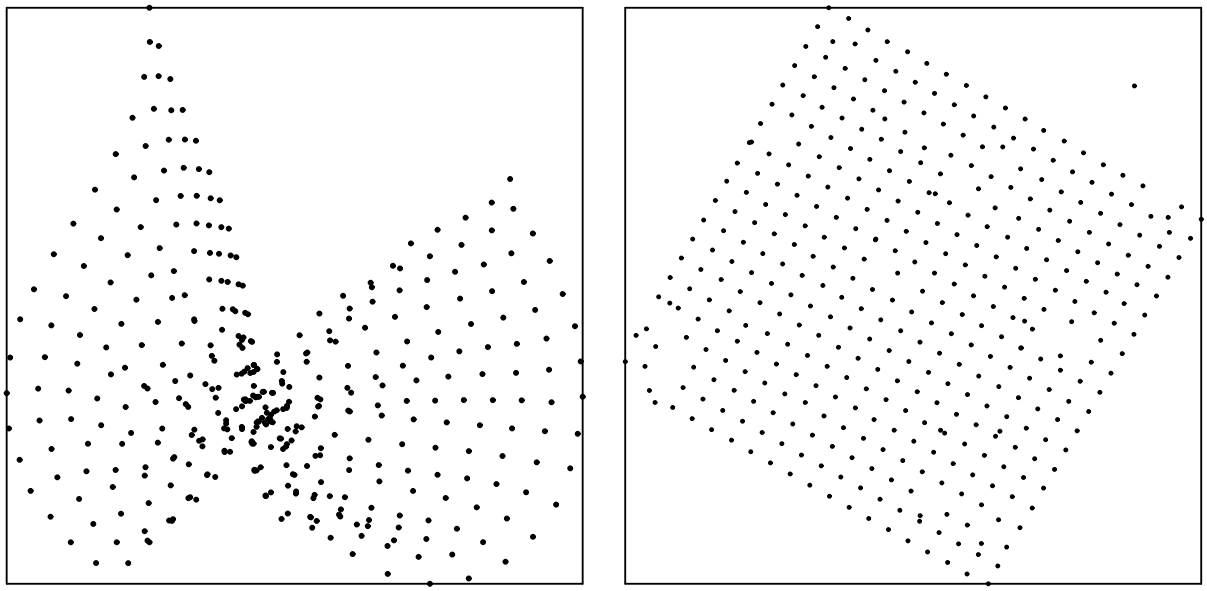


Figure 3-6: A pathological case showing the possible effect of communication patterns on the chosen coordinates. In the first case, nodes only contact their four nearest neighbors, allowing the resulting coordinates to twist over long distances. In the second case, nodes contact distant nodes as well, improving the accuracy of the coordinates at the larger scale.

probability p, to communicate with a faraway neighbor; otherwise the node will communicate with a local neighbor. Once the set has been chosen (far or near), a specific node to sample is chosen from the set uniformly at random.

Figure 3-7 shows the effect of p on the final accuracy of the coordinates. When half of the communication is to distant nodes, coordinates converge quickly. Convergence slows as the proportion of distant nodes increases, but similarly accurate coordinates are eventually chosen for small proportions of distant nodes, suggesting that even when only 5% of the samples involve distant nodes, skewed coordinate placements like those in Figure 3-6 that result in large errors will be avoided.

3.3.4 Adapting to network changes

Because Vivaldi constantly recomputes coordinates, it naturally adapts to changes in the network. To test how well Vivaldi responds to changes we created a synthetic "Transit-Stub" topology of 100 hosts using the ITM tool [13]. The median latency of this data set is 100ms.

We used Vivaldi to find coordinates for the 100 hosts; Vivaldi found a 6-dimensional fit using 32 randomly chosen neighbors. We then changed the network topology by increasing the length of one of the stub's connection to the core by a factor of 10. Figure 3-8 shows the median of the absolute error predictions made by each node over time. Prior to time 100 seconds the nodes have stabilized and the median prediction error is around 15ms. At time 100s the topology is changed to include the much longer transit-stub link. Shortly after the change the median error rises (to 25ms) because the system is using coordinates chosen for the old configuration to predict latencies in the new configuration. As nodes continue probing, Vivaldi finds (by time 120s) a new set of coordinates that predict error well in the new configuration. The error is higher following the change in the network because the new configuration is not as easily embedded as the original configuration. An alternate

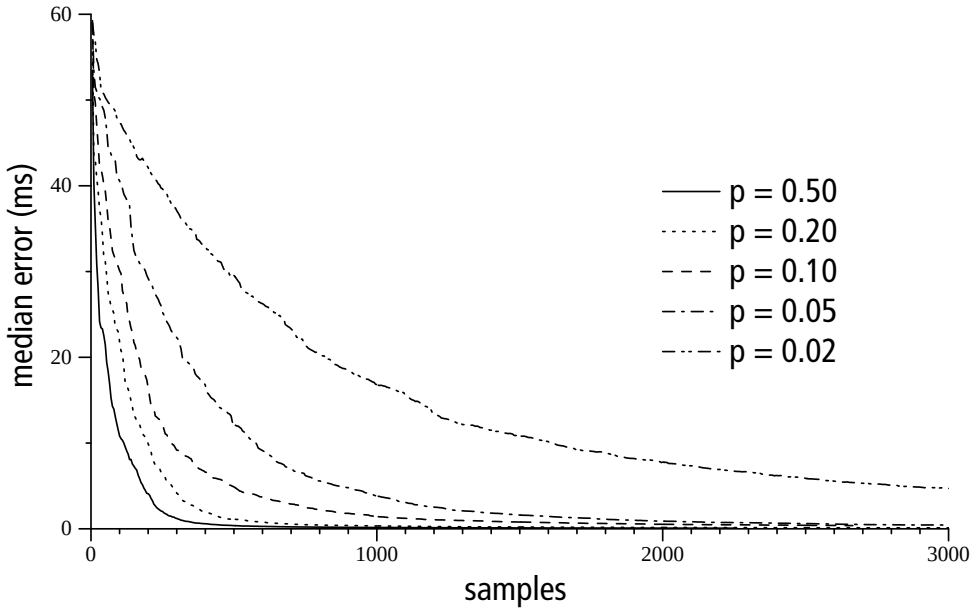


Figure 3-7: The effect of long-distance communication on the accuracy of the coordinates. Each line in the graph plots prediction error over time for an experiment in which nodes contact distant nodes (as opposed to nearby nodes) with probability p at each timestep.

explanation is that Vivaldi has failed to converge following the change. To show that the new configuration is, in fact, less embeddable, we restore the original configuration at time 300s. The nodes quickly reconverge to positions that give the same median error as the system prior to the topology change (around 15ms).

3.3.5 Network scale

Vivaldi also scales to large networks. The algorithm uses a constant amount of storage on each node, and can use measurements of traffic that would have been sent anyway as long as they conform to the patterns outlined in the previous section (i.e., at least 5 percent of the traffic be to distant nodes). The algorithm has no requirements that scale even logarithmically in the number of nodes, much less linearly.

The one exception is startup time for an initial network. A network of millions of nodes cannot coordinate itself from scratch in a constant amount of time. However, networks of millions of nodes tend not to spring up all at once. They start as smaller networks and grow incrementally. Once there is a critical mass of well-placed nodes in a Vivaldi network, a new node joining the system needs to make few measurements in order to find a good place for itself. That is, once there are enough nodes in the system, the joining cost for a new node is only a small constant number of network samples, regardless of the size of the network. New nodes converge quickly because they begin with a large initial time-step.

To demonstrate this claim, we initialized a 1,000-node network using the King data set. Once the network converged, we added 1,000 new nodes, one at a time, measuring the actual (not estimated) prediction error of each newly placed node as a function of the number of samples obtained each node obtains (equivalent to the number of other nodes each node communicates with). Each new node's prediction error is as low as it will ever be after about 20 samples.

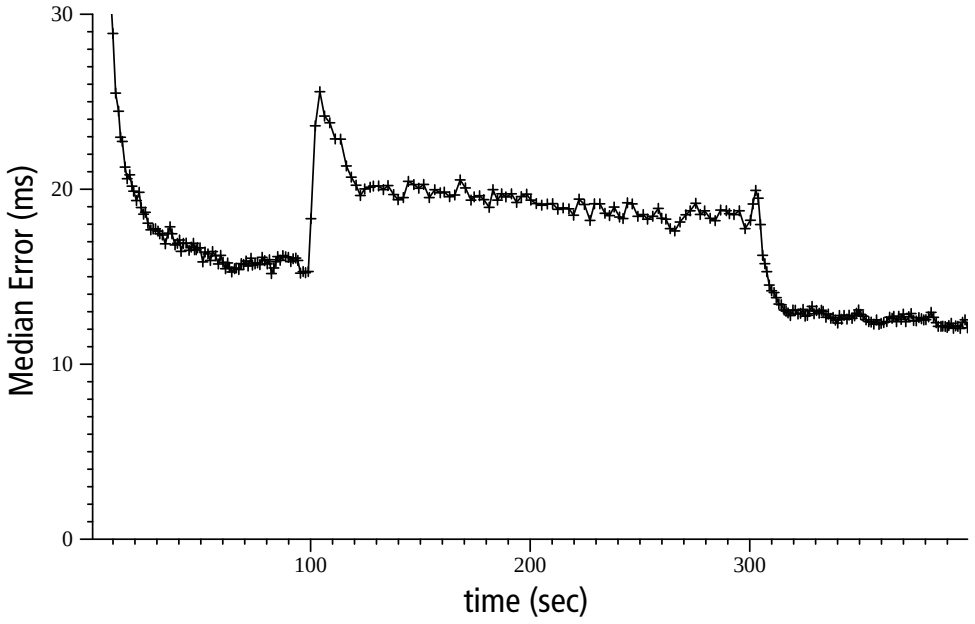


Figure 3-8: Vivaldi is able to adapt to changes in the network. In this experiment, we constructed a 100 node GTITM topology and allowed Vivaldi to determine coordinates for the nodes. The median error is plotted above against time. At time 100 one of the transit stub links is made 10 time larger; after around 20 seconds the system has reconverged to new coordinates. The error of the new system is larger in the original configuration. At time 300 the link goes back to its normal size and the system quickly reconverges to the original error.

3.3.6 Accuracy

To evaluate Vivaldi's accuracy, we compared it against GNP [88], a centralized algorithm, on the PlanetLab and King data sets. Figure 3-9 compares the cumulative distribution of prediction error for the 2-dimensional Euclidean coordinates chosen by Vivaldi and GNP for both the PlanetLab and King data sets. We configured GNP to use 32 landmarks in these experiments and limited each Vivaldi node to communication with 32 neighbors. Vivaldi's error is competitive with that of GNP. Also note that both systems are able to make good predictions: the median relative error of both systems is around 12 percent on the King data set.

In Section 3.3.3 we discussed how Vivaldi can avoid "folding" the coordinate space by communicating with some distant nodes. We also find that neighbor selection affects accuracy in another way: preferentially collecting RTT samples from some nodes that are nearby in the network improves prediction accuracy. This was first demonstrated by PIC [28] and likely occurs because the relative error metric we use to evaluate the embedding penalizes errors on short links more than on long links; preferentially measuring some nearby nodes improves the prediction to nearby nodes and reduces the error on short links. In the experiments presented in this section, each Vivaldi node took measurements from 16 nearby neighbors (found using the simulator's global knowledge of the network) and 16 random neighbors. Because GNP's performance depends on the choice of landmarks in the network, we performed 64 GNP experiments with random landmark sets and chose the set that gave the lowest median error.

The number of neighbors also affects the accuracy of Vivaldi. Figure 3-10 shows the distribution of RTT prediction errors for varying numbers of neighbors using 3-dimensional coordinates. The neighbors were chosen using half nearby neighbors as described above. Vivaldi's performance increases rapidly until about 32 neighbors, after which time it does

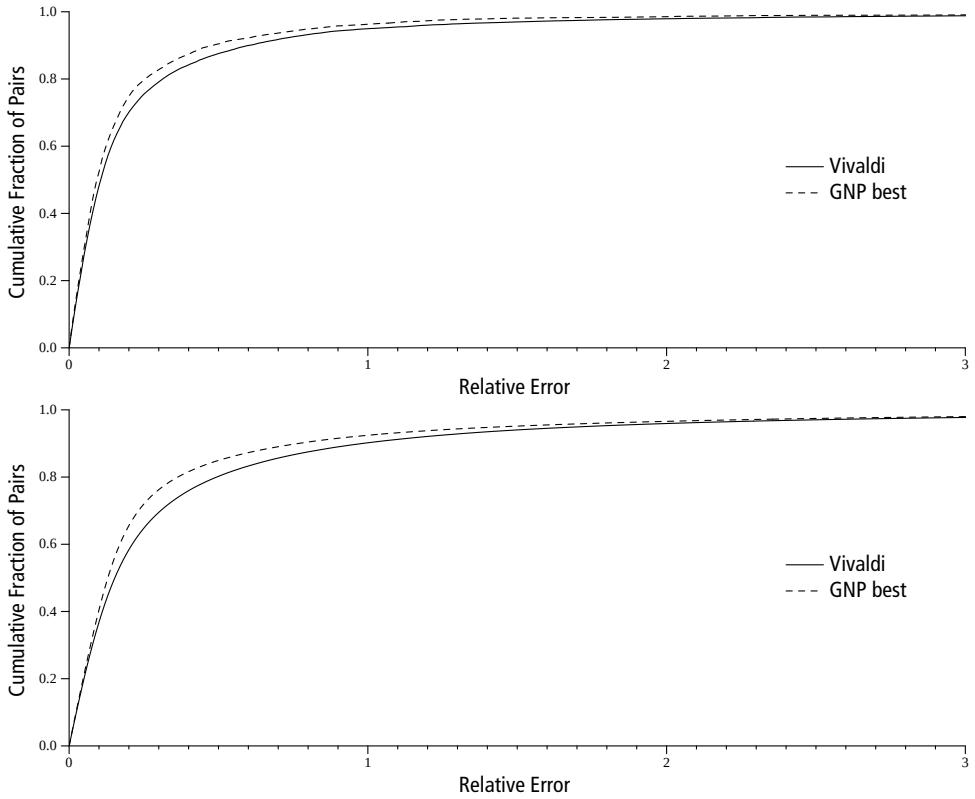


Figure 3-9: The cumulative distribution of prediction error for 2-dimensional Euclidean coordinates chosen by Vivaldi and GNP on the PlanetLab data set (top) and King data set (bottom).

not improve much. GNP requires fewer neighbors than Vivaldi (it works well with around 16), but Vivaldi is less sensitive to neighbor placement and can use any node in the system as a neighbor.

3.4 Model selection

Vivaldi was inspired by analogy to a real-world (and thus three-dimensional Euclidean) mass-spring system. As discussed in Section 3.1, the algorithm can use other coordinate spaces by redefining the coordinate subtraction, vector norm, and scalar multiplication operations. In this section, we consider a few possible coordinate spaces that might better capture the Internet's underlying structure for the King and PlanetLab data sets.

3.4.1 Triangle inequalities

Before considering any specific coordinate spaces, let us first consider how well we can expect to do. Almost any coordinate space we might consider satisfies the triangle inequality, which states that the distance directly between two nodes A and C should should be less than or equal to the distance along a path detouring from A to B and then to C. One should only expect to be able to find a coordinate space consistent with a set of inter-node latencies if the latencies themselves satisfy the triangle inequality.

Figure 3-11 presents an evaluation of the extent of such violations in the Internet. For each pair of nodes, we found the lowest-RTT path through any intermediate node and calculated

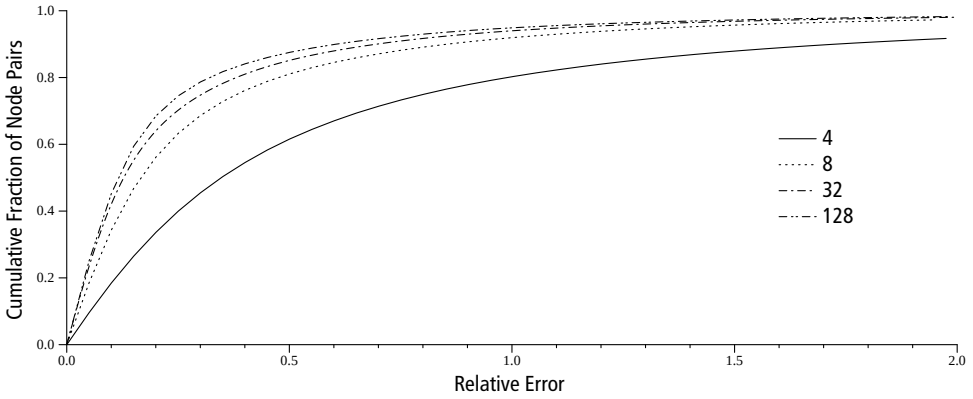


Figure 3-10: The cumulative distribution of prediction error for 3-dimensional coordinates chosen by Vivaldi using different numbers of neighbors.

the ratio of the RTTs of the indirect and direct paths. Figure 3-11 plots the cumulative distribution of these ratios for the PlanetLab and King data sets. The vast majority of node pairs in the King data set are part of a triangle violation: these small violations are due mainly to measurement inaccuracy. A smaller number of severe violations are present in both datasets, as well as in the analysis of Tang and Crovella [127]. Because only around five percent of node pairs have a significantly shorter two-hop path, we expect that both data sets will be "embeddable" in a Euclidean space that assumes no triangle violations. Were more node pairs in violation we would not expect Vivaldi to be able to find coordinates that predict latency well.

We also count the number of triples (i, j, k) that violate the constraint $|x_i - x_j| + |x_k - x_j| > |x_i - x_j| + \epsilon$. We include the constant term ϵ to avoid counting marginal violations (the error in measuring these links is likely on the order of several milliseconds). For $\epsilon = 5ms$, we find that 4.5% of the triples in the King data set violate the triangle inequality.

Of course, while a data set with many triangle inequalities will do poorly in most coordinate spaces we'd consider, a data set with few triangle inequalities still might not fit well into arbitrary coordinate spaces. The small number of triangle inequalities in the data only suggests that we are not doomed from the start.

3.4.2 Euclidean spaces

First, we explore the use of Euclidean coordinate spaces. These have the familiar equations:

$$\begin{aligned} [x_1, \cdots, x_n] - [y_1, \cdots, y_n] &= [x_1 - y_1, \cdots, x_n - y_n] \\ \|[x_1, \cdots, x_n]\| &= \sqrt{x_1^2 + \cdots + x_n^2} \\ \alpha \times [x_1, \cdots, x_n] &= [\alpha x_1, \cdots, \alpha x_n] \end{aligned}$$

If we choose to use a Euclidean coordinate space, the first question is how many dimensions to use. We can use a principal components analysis as in Cox and Dabek [30] and Tang and Crovella [127], to characterize the dimensionality of Internet coordinates under the assumption that they are a Euclidean space.

Principal components analysis is used to find the most "important" directions in a data set. The result of PCA is an ordered and ranked list of d directions (vectors) in a d dimensional

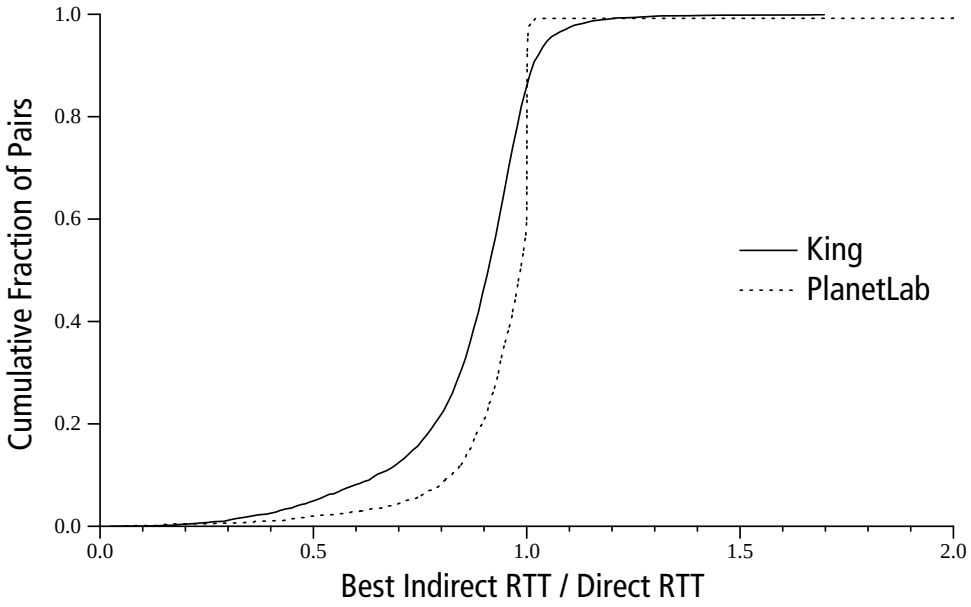


Figure 3-11: The cumulative distribution of the ratios of the RTTs of the best two-hop path to the direct path between each pair of King and PlanetLab nodes. The best indirect path usually has lower RTT than the direct path.

data set. In two dimensions, the first principal component is the line which minimizes the perpendicular distance to the points in the data set; this line points along the direction with the maximum variance in the points. The second principal component is orthogonal to the first; the difference in the rank of the two components reflects the difference in variance in the two directions.

By examining the direction and rank of each principal component we can gain insight into how much each additional dimension helps reconstruct the original coordinates. Figure 3-12 shows a simple example. The points in the figure are drawn from a two-dimension Gaussian distribution. We then applied PCA to find the two principal components. As we expect, the two components lie along the axes of the cluster of points. The vectors are scaled according the rank found by PCA; the axis with the larger variance produces a larger rank (in this example, we can also think of PCA as recovering the covariance matrix used to produce the samples). The directions of the vectors can be used for classification; we will mainly be interested in the lengths of the vectors. In this simple example, the ratio of the larger rank to the smaller (about 5:1) gives us a feeling for how "one-dimensional" the data is. The larger the ratio the less information is contained in the second dimension and the less improvement in prediction error we could expect in moving to a two-dimensional model from a one-dimensional one.

To apply this analysis to the Internet we perform PCA on the latency data in the King data set by first finding 10-dimensional Euclidean coordinates for each node using Vivaldi. Assuming that the Internet is less than 10-dimensional, we can use PCA to determine the relative importance of each output dimension. We'll be looking at the relative ranks of the eigenvalues produced by PCA. For example, if the first three are large and the rest small, we can conclude that a 3-dimensional model is well suited to the Internet.

Figure 3-13(b) shows a 2-dimensional projection of the 10-dimensional coordinates found by Vivaldi for the King data set. This projection is onto the plane defined by the first two principal components. Examining the data reveals that the latencies in the data are

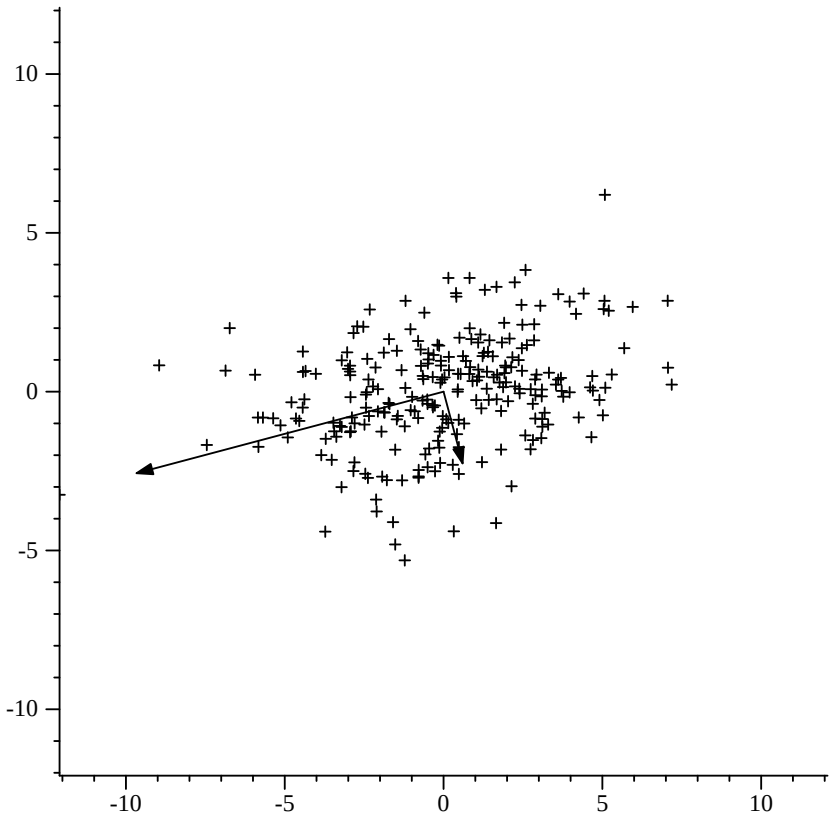


Figure 3-12: PCA example on points drawn from a 2-dimensional Gaussian distribution. The principal components are shown as vectors scaled to match their relative importance.

dominated by geographic distance (the clusters in Figure 3-13(b) correspond to the geographic regions). This finding is a somewhat surprising result given the complexity of the Internet but explainable by the relative sizes of the speed of light and the size of the Earth. The Earth is big enough (or light is slow enough) that the time required to move a signal from one point on the Earth's surface to another is a major contributor to latency. It also suggests that the data may be low-dimensional: if geographic distance were the only factor in latency, a 2-dimensional model would be sufficient. A 2-dimensional fit is not perfect, probably due to aspects of the network like access-link delays and the fact that nodes often take inefficient routes as they move from one backbone to another (as happens, for example, in hot-potato routing). As we add more dimensions, the accuracy of the fit improves slightly, probably because the extra dimensions allow Vivaldi more "wiggle room" for placing hard-to-fit points.

Figure 3-13(a) plots the eigenvalues generated by the PCA analysis of the King data set. This plot is called a scree plot after the piles of rock that heap up next to mountains: the mountain in this analogy is the peak formed by the large eigenvalues at left and the scree are the remaining, lesser ranked dimensions. Reading these plots involves making a judgment about where the "mountain" ends and the "scree" begins; the value is our estimate for the dimensionality of the data set. From this figure we can estimate that coordinates primarily use two to three dimensions, with little variation in the others. We draw this conclusion based on the small value of the eigenvalues corresponding to dimensions past the third. That is, insofar as the data is Euclidean, it is only two- or three-dimensional.

Figure 3-14 plots the CDF of relative errors for 2-, 3-, and 5-dimensional Euclidean

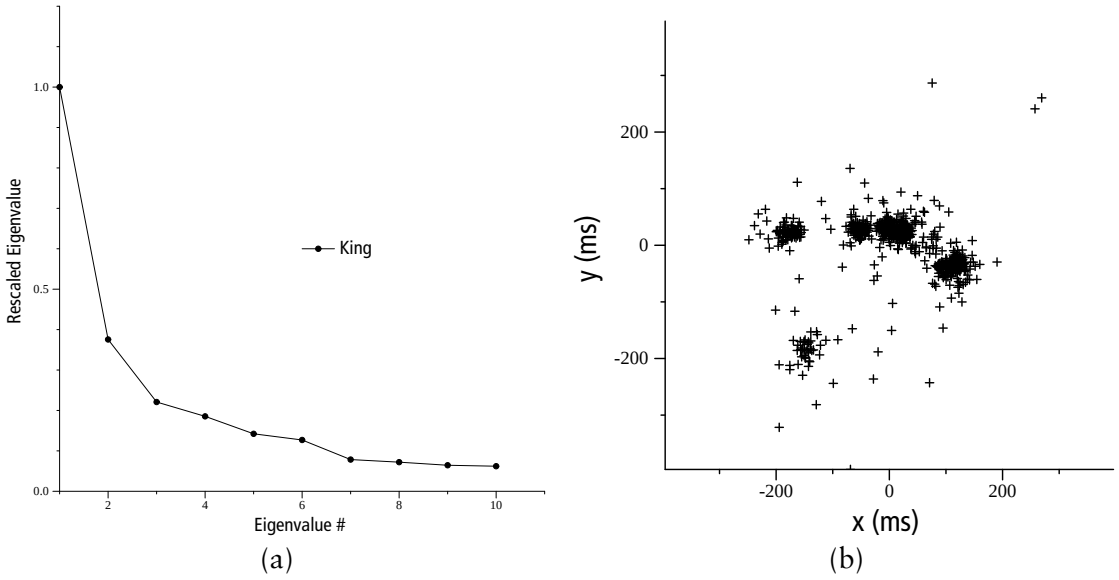


Figure 3-13: Scree plot of the eigenvalues produced by PCA for the King dataset (a) and the projection of the 10-dimensional coordinates found by Vivaldi for the data set projected so that the two largest principal components lie in the page (b).

coordinates, for both the PlanetLab and King data sets. Adding extra dimensions past three does not make a significant improvement in the fit. This result confirms our PCA analysis. While using coordinates with more dimensions does improve the quality of the fit, it also increases the communication overhead required to run Vivaldi; for this reason we prefer the lowest dimensional coordinates that allow for accurate predictions.

3.4.3 Spherical coordinates

Because we know that the distances we are attempting to model are drawn from paths along the surface of a sphere (namely the Earth), we might expect that a spherical distance function would provide a more accurate model.

We adapted Vivaldi to use spherical coordinates; instead of expressing a force vector in Euclidean space, a node moves some fraction of the angular distance towards or away from another node.

Figure 3-15 plots the CDF of relative errors for Vivaldi's computed 2-dimensional Euclidean coordinates as well as spherical coordinates with varying radii, for the PlanetLab and King data sets and a synthetic data set generated from a spherical model. In (a) and (b), the error is large until about 80 ms; before this, the sphere's surface is likely too small for the nodes to spread out.

The spherical model's error is similar to the Euclidean model's error and does not decrease as the radius is increased. In larger spheres, all points cluster on one side of the sphere as the sphere provides approximately $2\pi r^2$ of surface area, approximating a Euclidean plane in which to work. To test this hypothesis we tried Vivaldi on a synthetic network of nodes chosen to fit a sphere of radius 80 ms. The error for the spherical fit is zero when the radius of the modeled sphere is 80ms. The error increases as the radius increases beyond its optimal value. A fit to a 2-dimensional Euclidean space produced a larger error than the (near-perfect) spherical model on this data set.

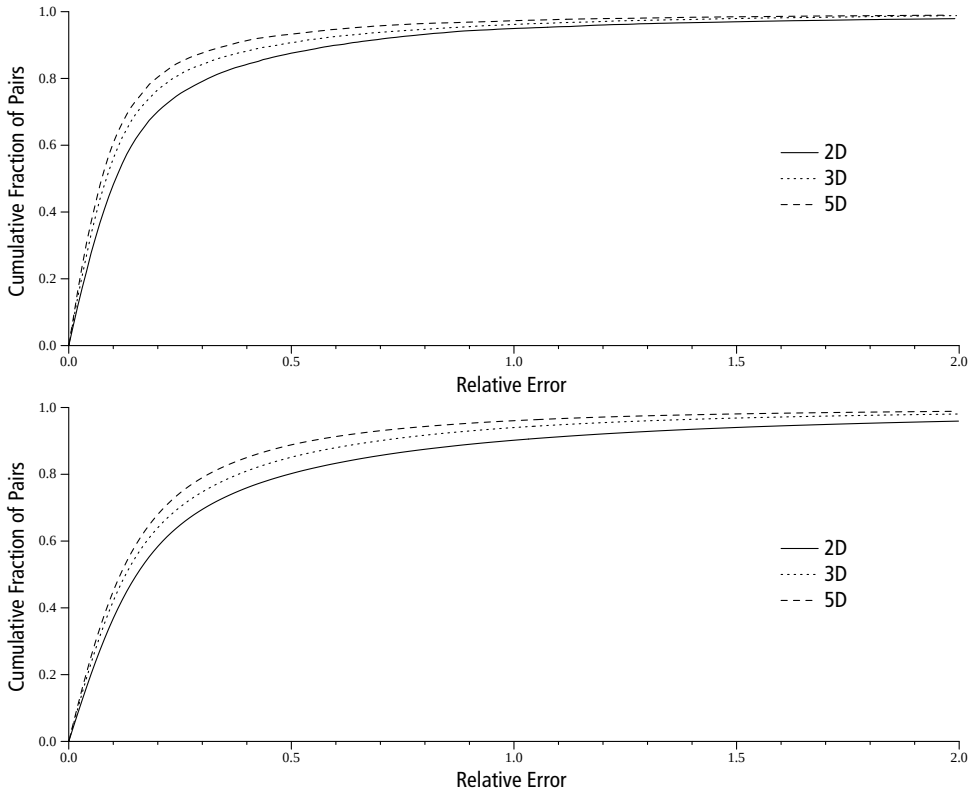


Figure 3-14: The cumulative distribution of Vivaldi's prediction error for various numbers of Euclidean dimensions for the PlanetLab (top) and King (bottom) data sets.

We suspect that the underlying reason that spherical coordinates do not model the Internet well is that the paths through the Internet do not "wrap around" the Earth appreciably. Inspection, using traceroute, of Internet paths originating in east Asia suggests that few links connect Asia and Europe directly. For instance, packets sent from Korea to Israel travel east across both the Pacific and Atlantic oceans rather than west across land. Some paths do connect Asia and Europe directly, of course, but they are not prevalent in the data. A spherical model assumes that such links would always be used when they make the path shorter. Since this case is not the usual one, the fact that the spherical model correctly predicts the few paths across Asia is negated by the fact that it incorrectly predicts the many paths that go the long way, avoiding Asia.

3.4.4 Height vectors

Spherical coordinates do not model the Internet well, but we devised a new model that attempts to capture what we know about the way the Internet is structured: a height vector consists of a Euclidean coordinate augmented with a height. The Euclidean portion models a high-speed Internet core with latencies proportional to geographic distance, while the height models the time it takes packets to travel the access link from the node to the core. The cause of the access link latency may be queuing delay (as in the case of an oversubscribed cable line), low bandwidth (as in the case of DSL, cable modems, or telephone modems), or even the sheer length of the link (as in the case of long-distance fiber-optic cables).

A packet sent from one node to another must travel the source node's height, then travel

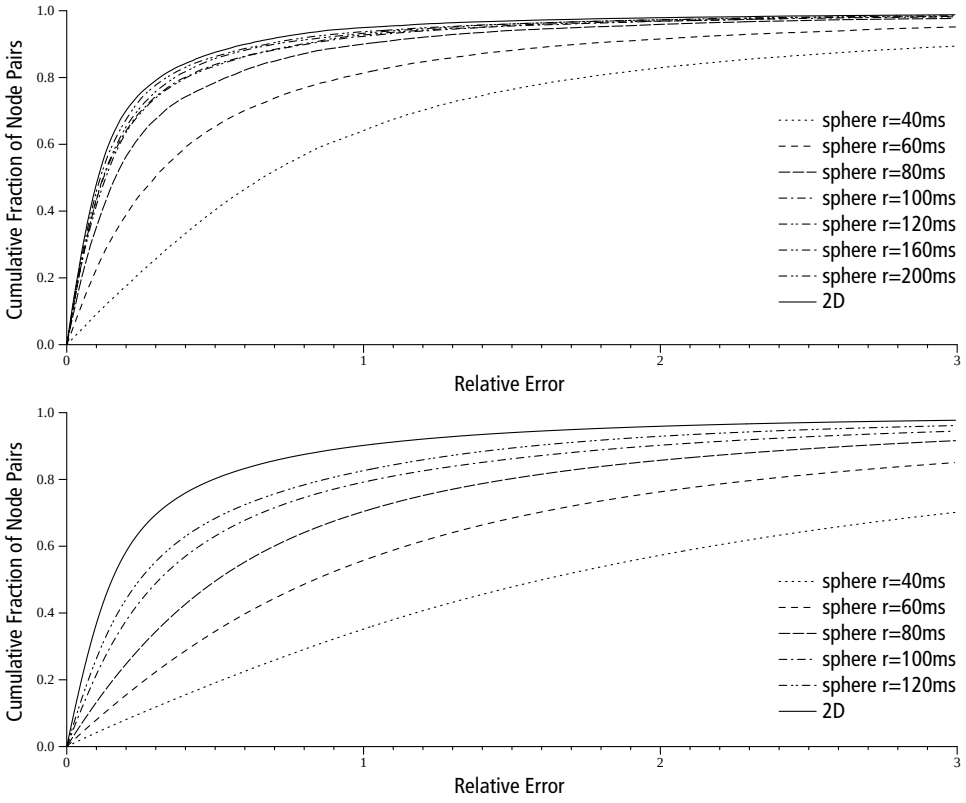


Figure 3-15: The cumulative distribution of prediction error for spherical coordinates of various radii chosen by Vivaldi for the PlanetLab (top) and King (bottom) data sets.

in the Euclidean space, then travel the destination node's height. Even if the two nodes have the same height, the distance between them is their Euclidean distance plus the two heights. The fact that, by adjusting its height, a node can move away from (or towards) all other nodes makes the height vector model different from adding a dimension to the Euclidean space. Intuitively, packet transmission can only be done in the core, not above it.

The height vector model is implemented by redefining the usual vector operations (note the $+$ on the right hand side of the subtraction equation):

$$\begin{aligned}
[x, x_h] - [y, y_h] &= [(x - y), x_h + y_h] \\
[x, x_h] + [y, y_h] &= [(x + y), x_h + y_h] \\
\left\| [x, x_h] \right\| &= \left\| x \right\| + x_h \\
\alpha \times [x, x_h] &= [\alpha x, \alpha x_h]
\end{aligned}$$

Node heights are restricted to being positive.

The effect of these equations is the following. In a normal Euclidean space, a node that finds itself too close to another node will move away from the other node. A node that finds itself too close to nodes on all sides has nowhere to go: the spring forces cancel out and it remains where it is. In the height vector system, the forces cancel out in the Euclidean plane, but the height forces reinforce each other, pushing the node up away from the Euclidean plane. Similarly, a node that finds itself too far away from other nodes on all sides will move down closer to the plane.

```
// Node i has measured node j to be rtt ms away,
// and node j says it has coordinates x_j.
simple_vivaldi(rtt, x_j)
  // Compute error of this sample. (1)
  e = rtt − ||x_i − x_j||
  // Find the direction of the force the error is causing. (2)
  dir = u(x_i − x_j)
  // The force vector is proportional to the error (3)
  f = dir × e
  // Flip the sign on the height portion of the vector (3.5)
  f.ht = −f.ht
  // Move a a small step in the direction of the force. (4)
  x_i = x_i + δ × dir
```

Figure 3-16: The simple Vivaldi algorithm, showing the modification (line 3.5) in the force computation necessary to use height-vector coordinates.

Vivaldi has the option of correcting height or position in the plane when using this height-vectors. Our definition for the operations causes Vivaldi to evenly divide the error between node heights and positions in the plane. Also note that one additional step must be taken when computing force vectors using the height-vector model. The force vector is the direction the node will move to update its coordinates. If node a measures its latency to node b and finds that it is too far from b in the coordinate space, the force vector in the plane will point from a to b since subtraction defines the direction of the vector in the plane. Using the results of our subtraction operation (which adds the heights) directly, however, would cause the two nodes to both move away from the plane: this is the opposite of the desired effect. Following the calculation of the force vector, we must invert the sign on the height portion of the vector. The same inversion happens if the nodes are two close together: in this case the initial height portion of the force vector is negative (since $\left(rtt - \|x_i - x_j\|\right)$ is negative); inverting makes the nodes move up in the plane as we expect. Figure 3-16 shows the updated pseudocode for calculating forces using height vectors; line 3.5 is the only change from the pseudocode shown in Figure 3-2.

Figure 3-17 shows that height vectors perform better than both 2D and 3D Euclidean coordinates. The effect of height vectors is most noticeable in the tails of the distribution: using height vectors instead of 3-dimensional coordinates reduces the 75th percentile error from 31 percent to 26 percent.

Examination of the coordinates that the height vector model assigns to hosts in the PlanetLab data set shows that the model captures the effect we hoped. Well-connected nodes, such as the ones at New York University, are assigned the minimum height. Two Brazilian nodes are assigned coordinates at approximately 95 ms above the United States. Using traceroute on these nodes we see that 95 ms is approximately the distance to the nodes' connection to the backbone of the Internet2 network inside the United States. Because the Brazilian nodes send their traffic via the United States to other nodes in this small data set they are best represented as 95 ms "above" the continental U.S. If there were a larger variety of Brazilian nodes and links in the data set, the nodes would be given their own region of the Euclidean plane.

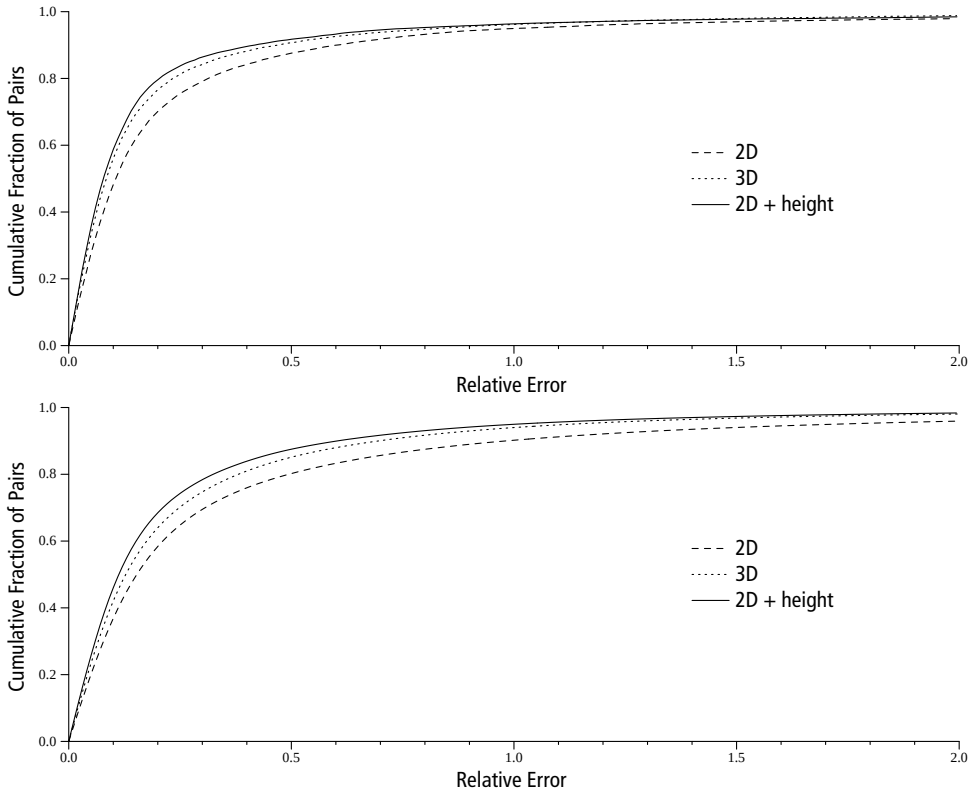


Figure 3-17: The cumulative distribution of prediction error for 2- and 3-dimensional Euclidean coordinates and height vectors chosen by Vivaldi for the PlanetLab (top) and King (bottom) data sets.

3.4.5 Graphical comparison

The plots in Figure 3-18 show the final placement of nodes in two and three dimensional Euclidean space and with height vectors. Most of the nodes are part of four large clusters that correspond to geographic regions. The Euclidean plots, Figures 3-18 (a) and (b), show a collection of nodes that are between these clusters and don't seem to know where to go. These nodes are also the bulk of the ones with large errors. The height plots have a place for these nodes: high above the rest of the network. This results in a smaller maximum node error for the height plots, as well as a smaller median error. The smaller errors suggest that the height vectors are a more accurate reflection of the structure of the Internet for the nodes in the data sets.

3.5 Theoretical results

Previous sections demonstrated that the coordinates produced by Vivaldi make good predictions for latencies on the Internet. Here we summarize some theoretical results that help us understand how well an algorithm like Vivaldi can perform and how difficult the problem of finding coordinates is. Much of this work was published after the initial GNP and Vivaldi publications [90, 31, 33], and in some cases was explicitly inspired by GNP and Vivaldi.

Vivaldi can be characterized as an embedding problem: it embeds the non-metric space defined by Internet latencies into an n-dimensional Euclidean space. Results for the embedding of metric spaces onto low-dimensional spaces can help us understand how well Vivaldi

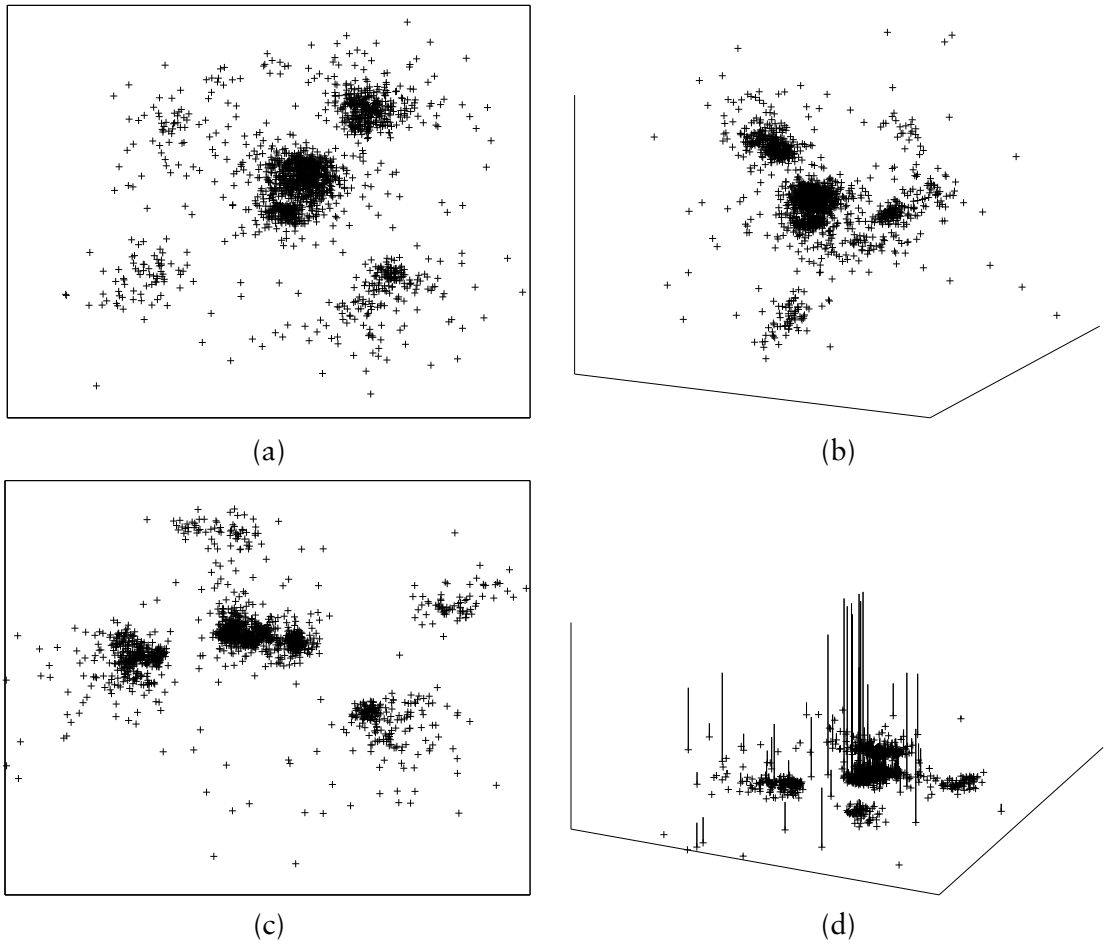


Figure 3-18: The node placement chosen by Vivaldi for the King data set (a) in two dimensions, (b) in three dimensions, (c) with height vectors projected onto the xy plane, and (d) with height vectors rotated to show the heights. The "fringe nodes" in (a) and (b) are not as prevalent in (c).

could possibly do. These results have limited applicability, however, because they apply only to metric spaces (Internet latencies are not a metric space since the triangle inequality is violated) and because they consider the worst relative error among the predicted links (applications that use Vivaldi are more likely to be interested in the median error). Finally, many of these results assume that all of the pair-wise distances in the metric space are available. In the context of Vivaldi, this assumption is equivalent to finding coordinates given the matrix of latencies (if one had the matrix of latencies, latency prediction would not be necessary).

Linial et al. analyze the distortion that results from embedding a metric space (derived from distances in a graph) into a low-dimensional space [74]. An embedding causes a distortion of c if all distances in the original space are expanded or contracted by less than a factor c; this is roughly equivalent to saying the relative error of every predicted latency is less than c. Their analysis suggests that it is possible to embed a metric space into a space with the Euclidean distance function (l_2) using $O(\log N)$ dimensional coordinates with a distortion of $O(\log N)$. A high-dimensional embedding can be found in random polynomial time by using semidefinite programming, a method of solving optimization problems that is related to, but more general than, linear programming. The embedding found by semidefinite programming can be reduced to $O(\log N)$ dimensions by projection into a random, low-dimensional

($O(\log N)$) subspace. This result gives us some hope that Vivaldi will be able to find an embedding, but since distortion is essentially worst case relative error this result doesn't tell us much about the quality of the embedding we should expect from Vivaldi. A $O(\log N)$ distortion is very large compared to the median error achieved by Vivaldi (equivalent to distortion slightly more than 1).

Kleinberg et al. analyze a problem more directly related to Vivaldi [66] by considering algorithms that do not have access to the distance matrix but can only measure the distance to a set of beacon nodes in the system and then must predict distances between any pair of nodes in the system. The analysis introduces the notion of slack: some fraction of distances (the slack, or, ϵ) can be arbitrarily distorted by the embedding. Under the assumption that the Internet is a doubling metric [63], Kleinberg shows that an algorithm that measures only the distance to a set of beacons can produce an embedding that has nearly no distortion for $1-\epsilon$ of the distances. Slivkins [119] continues this analysis and shows that a fully distributed algorithm (in the same sense that Vivaldi is fully distributed: every node performs the same amount of computation) exists to find a low-distortion embedding with slack for Internet latencies. Both of these algorithms use triangulation to beacons ("virtual beacons" in the latter algorithm) to bound unmeasured distances.

These algorithms are not directly comparable to Vivaldi or GNP (they assume that the triangle inequality always holds, for instance, when it is, in practice, violated for a small fraction of node triples), but they provide an alternative explanation for why Vivaldi and GNP are able to find coordinates that predict latencies well.

The use of slack is important: if distortion (without slack) is considered, it is impossible to embed a doubling metric into Euclidean space with constant distortion [52]. Any embedding that will be used in practice to make predictions should have constant distortion (ideally 1). Although an algorithm that embeds using slack is likely to have a large distortion, it will still be useful to applications as long as most predictions are accurate. Vivaldi and other coordinate algorithms produce bad predictions for some fraction of the links in the system and have still proven useful.

Little is known about the running time of finding good embeddings. The problem of approximating an exact embedding of a metric space onto a line (1-dimensional space) is known to be NP-hard [117]; this is a lower-dimensional version of the embedding that Vivaldi performs. The difficulty of more general embeddings is an open problem [79].

4

Latency

This chapter investigates design choices that affect DHT `get` latency. We'll consider the latency in the context of a static network (that is, a network where nodes do not fail); this analysis then represents the best latencies that DHash can achieve given the distribution of Internet round-trip times. In an environment where nodes fail, `get` latency will increase because RPCs sent to failed nodes delay the operation by a timeout interval. The main contribution of this chapter is an understanding of why one optimization, proximity neighbor selection, performs as well as it does.

Recall from Chapter 1 the operation of the naive algorithm against which the optimizations described in this chapter will be judged, called base DHash. Each 8192-byte block is stored as 14 1171-byte erasure-coded fragments, any seven of which are sufficient to reconstruct the block. The 14 fragments are stored at the 14 immediate successors of the block's key. When an application calls `get(key)`, the originating node performs an iterative Chord lookup, which ends when the key's predecessor node returns the key's 16 successors; the originating node then sends seven parallel requests the first seven successors asking them each to return one fragment. We'll consider the `get` operation in two parts. The lookup part of the operation routes a message to the predecessor of the key and returns the locations (IP addresses) of the block's fragments. The fetch stage of the operation retrieves these fragments from the locations returned by the lookup.

The following sections will provide necessary background on the Chord protocol and discuss each of these optimizations in detail.

4.1 Chord background

As discussed in Section 1.1.2 Chord implements a hash-like lookup operation that maps 160-bit data keys to hosts: lookup(key) $\rightarrow$ (id, IP). Lookup accepts a key and returns the identifier and IP address of the node to which the key is mapped. DHash uses the lookup operation to map data blocks to nodes based on the key of the data block.

The challenge facing Chord is to perform this mapping accurately when the number of hosts in the system is very large and the system membership is highly dynamic. We assume that the system is too large or dynamic to allow each host to maintain an up-to-date list of all other nodes in the system. Instead, each Chord node maintains a small amount of state (on the order of $\log N$ entries per host where N is the number of nodes in the system) and a lookup request can only be completed with the help of a number of other nodes.

Using the successor pointers that each node maintains (as described in Section 1.1.2), a node could perform a lookup by forwarding messages through successor pointers: using this algorithm, a lookup would require a number of hops that grows linearly with the number

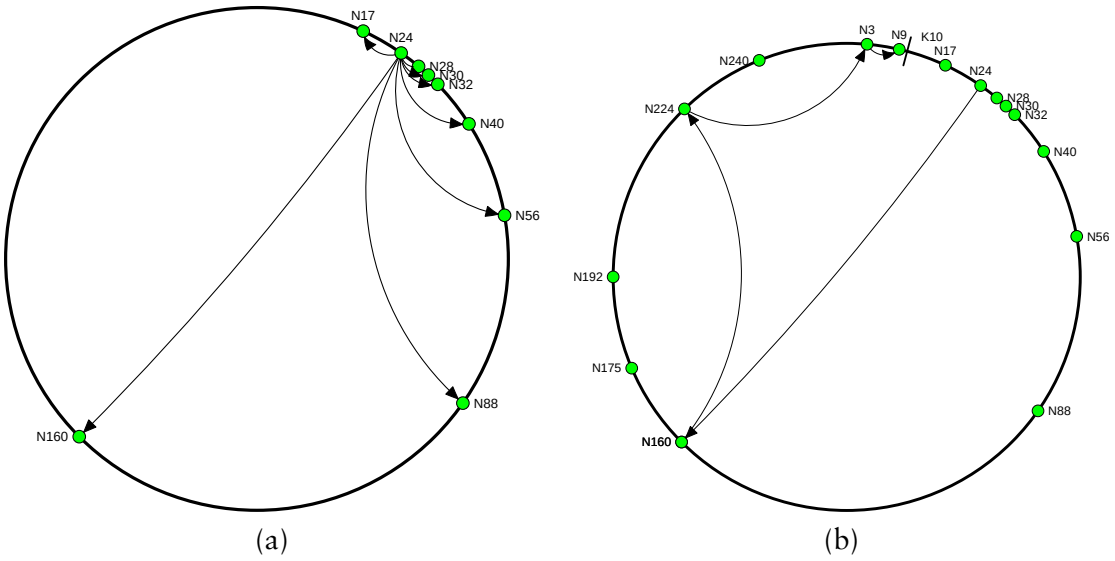


Figure 4-1: (a) A Chord node's finger table. Each node maintains $\log N$ pointers to other nodes. The pointers are spaced exponentially around the ring (i.e. $\frac{1}{2}, \frac{1}{4}, \frac{1}{8} \ldots$ of the way around the ring). In the example above, node N28's routing table is shown. The most distant finger of node 28 points to the first node that is more than half-way around the ring (after $28 + 256/2 = 156$); this is node N160 in the example. This spacing allows a lookup to halve the distance to the target at each step. Lookups complete in $\log N$ time. (b) The path of a lookup for key K10 originated at node N24.

of nodes in the system. To reduce the number of hops in a lookup, Chord nodes maintain additional information about other nodes in the form of a routing table, called the finger table, that allows the node to perform lookups in $O(\log N)$ time. The finger table for a node n contains $\log N$ entries that point to hosts at power-of-two distances ahead of n along the identifier circle: the most distant finger points at the first host whose ID is half-way around the ring from n's ID, the next finger $\frac{1}{4}$ of the way, and so on. Figure 4-1(a) shows an example of a finger table.

As shown in Figure 4-2, to find a key's successor using the finger table, a node routes the request greedily in ID space. At each step, the requesting node consults the node that is closest to the key's ID but still precedes the key on the ring. That node replies with the identity of the best next hop node. Eventually the requesting node will contact the key's predecessor, which will return its successor list: the answer to the lookup query. Because of the power-of-two distribution of a node's finger table, the node will always have a pointer that is at least half of the distance to the key at each step in the lookup. Because the distance remaining to the key is halved at each step of the lookup, we expect a lookup to require $O(\log N)$ hops. Figure 4-1(b) shows the path taken by an example lookup.

We can more precisely bound the number of hops taken by a lookup: a lookup will take $\frac{1}{2} \times \log N$ hops on average. To see this, consider that each step of the lookup contacts a node that has one more bit in common with the key than the destination. The lookup can correct at most the high $\log N$ bits before it reaches the successor node. If the source node and key are random (as they will be in the experiments in this section), on average half of the high-bits will already be equal. Thus, $\frac{1}{2} \log N$ hops are required to correct the remaining bits.

An optimization to Chord that isn't explored in this thesis is increasing the amount of information that Chord keeps about other nodes in the system. One way to do this would be to change the base of the finger table. By keeping a larger finger table, each hop could move

```
// Ask node n to find id's successor; first
// finds id's predecessor, then asks that
// predecessor for its own successor.
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor_list();

// Ask node n to find id's predecessor.
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor()])
    l = n'.closest_predecessor(id);
  return n';

// Ask node n for the node in its finger table or
// successor list that most closely precedes id.
n.closest_predecessor(id)
  C = {n' ∈ {fingers ∪ successors} s.t. n' ∈ (n, id]}
  return argmin_{n∈C}(distance(n, id))
```

Figure 4-2: The pseudo-code to find the successor node of an identifier *id* using iterative lookup. Remote procedure calls are preceded by the remote node.

$\frac{3}{4}$ of the way to the target, for example, instead of half way. In general, by increasing the size of the routing table to $(b-1)\log_b N$ Chord can achieve $\log_b N$ hop lookups [62]. These optimizations would reduce latency under low churn, because each node would know about many other nodes. On the other hand, in high churn networks, these optimizations require more bandwidth to keep routing tables up to date and experience more timeouts because routing tables contain recently-failed nodes. See Li. et al. [72, 73] for a thorough treatment of this optimization.

We also don't consider the effect of caching data items on the lookup path. This optimization places a copy of fetched data blocks on one more nodes near the end of the lookup path. Because lookups from random nodes for the same key tend to converge as they near the predecessor of a key, these cached blocks are likely to be encountered by future lookups. This optimization effectively spreads the load of serving a popular block in proportion to its popularity: when caching is used, each node can expect to see no more than $\log N$ requests for the same block. Caching also reduces lookup latency by reducing the hop count of lookups. Block caching is considered in more detail in prior work [32].

4.2 Recursive or iterative?

The base Chord and Kademlia [80] algorithms are iterative; other lookup algorithms use recursive lookup [16, 139]. In the recursive style, the requesting node forwards the lookup to the first hop which forwards in turns forwards the request to the next best hop (instead of returning information about the next best hop to the requester). When the lookup reaches the predecessor, it sends a message to the node that originated the lookup with the results of the lookup. Figure 4-3 shows the pseudocode for recursive lookup using finger tables. Note

```
// ask node n to find the successor of id
// This lookup is being done on behalf of node orig
n.find_successor(id, orig)
   if (id ∈ (n, successor])
      orig.lookup_done(successor_list);
   else
      n' = closest_preceding_node(id);
      n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
   for i = m downto 1
      if (finger[i] ∈ (n, id))
         return finger[i];
   return n;

// called when a lookup completes. Return the results
// of the lookup to the user
n.lookup_done(successors)
```

Figure 4-3: Recursive Chord lookup using the finger table. find_successor returns the successors of key *id* by forwarding the lookup to the finger table entry that is closest to the target. Note that *finger*[0] is the node's successor.

that no messages are sent to the originating node prior to the lookup's completion; in iterative lookup (Figure 4-2) intermediate nodes send information about possible next hop nodes to the originating before the lookup can proceed. Recursive lookup might eliminate half the latency of each hop since each intermediate node can immediately forward the lookup before acknowledging the previous hop.

Figure 4-4 shows the effect of using recursive rather than iterative lookup in the simulator with the 2048-node King data set. For each technique, 20,000 lookups were performed, each from a random host for a random key. The average number of hops is 6.3. Recursive lookup takes on average 0.6 times as long as iterative. This decrease is not quite the expected factor of two: the difference is due to the extra one-way hop of (on average) 77 milliseconds to return the result to the originator.

While recursive lookup has lower latency than iterative, iterative is easier for a client to manage. If a recursive lookup elicits no response, the originator has no information about what went wrong and how to re-try in a way that is more likely to succeed. Sometimes a simple re-try may work, as in the case of lost packets. If the problem is that each successive node can talk to the next node, but that Internet routing anomalies prevent the last node from replying to the originator, then re-tries won't work because only the originator realizes a problem exists. In contrast, the originator knows which hop of an iterative lookup failed to respond, and can re-try that hop through a different node in the same region of the identifier space.

On the the other hand, recursive communication may make congestion control easier (that is, it is it may make it more feasible to rely on TCP). We will show in Chapter 5 that the performance of a naive TCP transport, even using recursive lookup, can be quite poor.

DHash uses recursive lookups by default since they are faster, but could fall back on iterative lookups after persistent failures. The current implementation can perform iterative

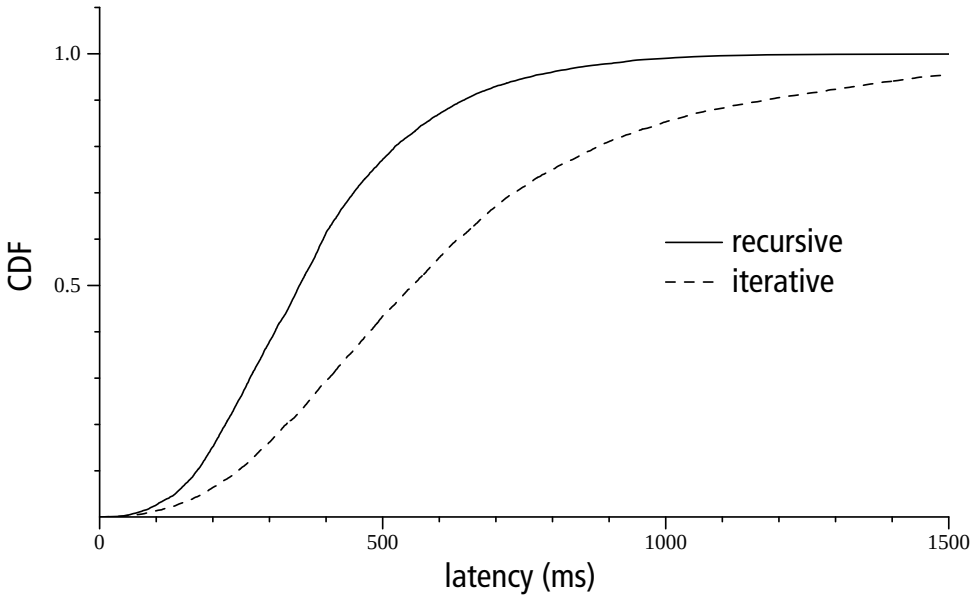


Figure 4-4: The cumulative distributions of lookup time for Chord with recursive and iterative lookup. The recursive median and average are 461 and 489 milliseconds; the iterative median and average are 720 and 822 milliseconds. The numbers are from simulations.

and recursive lookups but does not dynamically fall back to iterative lookup when a recursive lookup fails.

4.3 Effect of choice

The latency optimizations presented here rely on the observation that increasing the number of nodes an operation can choose from when forwarding a request decreases the latency of that operation by allowing the application to choose a nearby node. DHash attempts to make this set of nodes as large as possible at each step of a lookup request. In this section we will quantify how much the size of the set of available nodes affects the latency of the operation.

We are interested in the latency to the closest of n potential nodes. If the nodes are chosen at random, this latency is the minimum of n random samples from the distribution of round trip times. The distribution of this latency can be written as:

$$y_n(x) = n[1 - F(x)]^{n-1} f(x)$$

where $F(x)$ is the cumulative distribution function of the latencies and $f(x)$ is the probability distribution function (PDF). This expression can be understood as the probability that $n-1$ samples are greater than x and the remaining sample is x. Figure 4-5 shows this distribution for a number of sample sizes when sampling the King data set. The y-value of a point on the curve labeled "8 samples" is the probability that the minimum of 8 samples takes the corresponding x-value. As we expect, when more samples are considered the expected latency of the minimum is lower: the mass of the distribution moves to the left.

A useful summary of this distribution is the median of the minimum sample. We can compute the median by first finding the CDF for the above distribution:

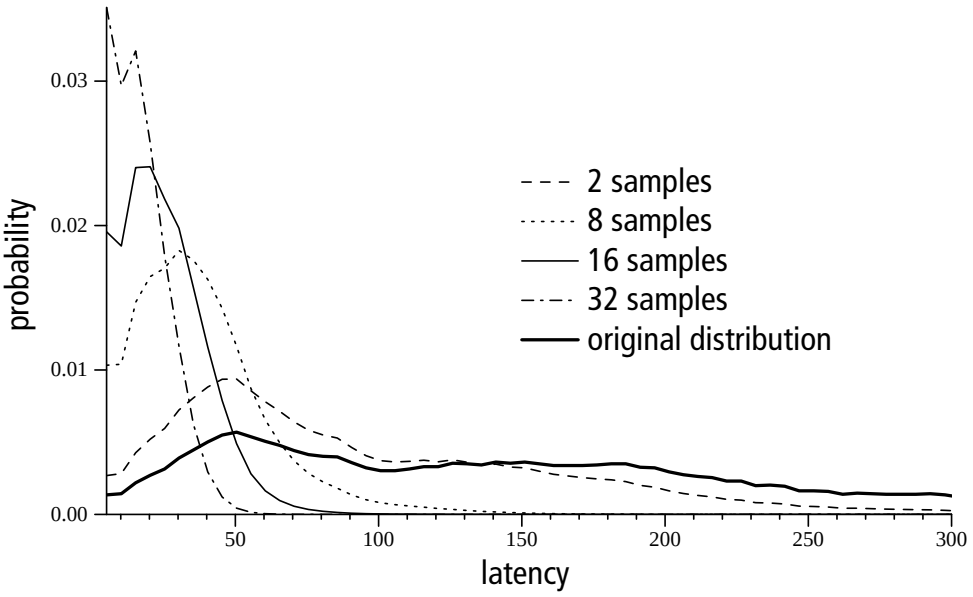


Figure 4-5: The distribution of the minimum sample latency for different sample sizes. Samples drawn from the distribution of latencies on the King data set are shown as the solid bold line. As a larger number of samples is considered, the minimum of that sample is more likely to be smaller. The distributions for larger sample sizes have more probability mass at smaller latencies (towards the left of the graph)

$$Y_n(x) = (1 - F(x))^n$$

This expression corresponds to the probability that all n samples are greater than x (the probability that 1 sample is greater than x is $1 - F(x)$). If we set this quantity equal to 0.5 and solve for x we can find the median value:

$$\begin{aligned} 0.5 &= (1 - F(x))^n \\ 0.5^{\frac{1}{n}} &= (1 - F(x)) \\ F(x) &= 1 - 0.5^{\frac{1}{n}} \end{aligned}$$

This tells us that the median of the distribution of the minimum element occurs at the value where the original distribution's CDF is equal to $1 - 0.5^{1/n}$. Another way of saying this is: the median of the minimum of n samples is at the $1 - 0.5^{1/n}$th percentile of the original distribution. A reasonable approximation for this value (found using Newton's method) is that the median falls at the $1/2x$th percentile for large x. If the distribution of latencies is uniform, doubling the sample size halves the minimum sampled latency.

Figure 4-6 shows the median for various sample sizes from the King data set. The dotted line is the CDF of the latencies between nodes in the King data set. Only part of the distribution (latencies less than 100ms) is shown. The boxes on the graph show how the number of samples influences the minimum of the sampled elements. Horizontal lines are labeled with the number of samples and are placed at the appropriate percentile (2 is at $1 - 0.5^{0.5} = .29$ for example). In this case, the x-value of the corresponding vertical line shows the median latency for the minimum of two samples (approximately 75ms).

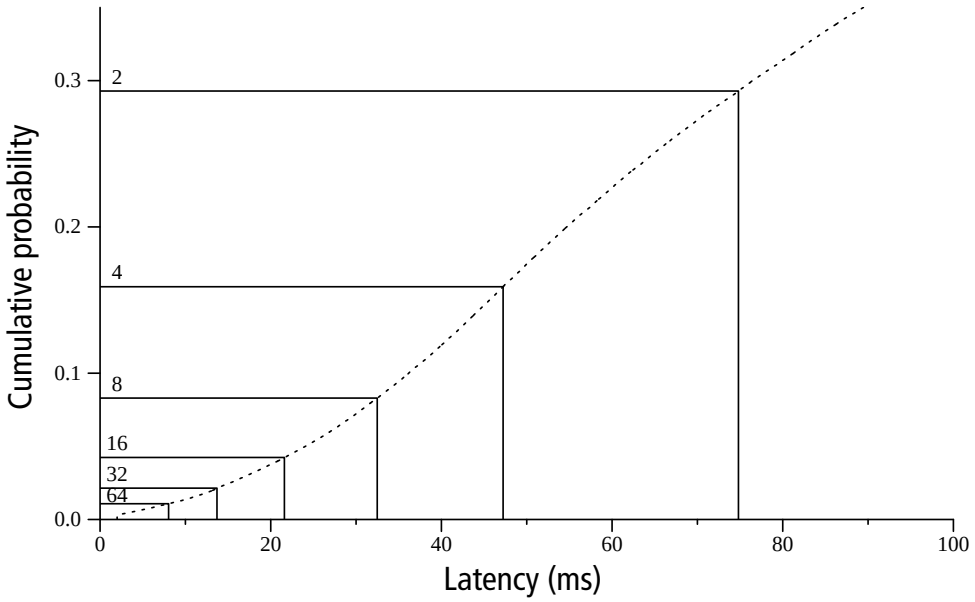


Figure 4-6: The median of the minimum latency of a sample of 2,4,8,16, and 32 samples drawn from the all-pairs latency distribution of the King dataset. The median sample latency for a given number of samples can be found by following the horizontal line labeled with that number of samples to its intersection with the CDF of the King latencies and down to the corresponding point on the x axis.

Given a DHT design that, like DHash stores blocks on randomly chosen servers, one can begin to form some expectations about fetch latency. The lower bound on the latency of a `get` operation is the round trip time from the originator to the nearest replica of the block, or the time to the most distant of the closest set of fragments required to reconstruct the block. For the typical block this time is determined by the distribution of inter-host delays in the Internet, and by the number of choices of replicas or fragments (as we have shown above). The DHT lookup required to find the replicas or fragments will add to this lower bound, as will mistakes in predicting which replica or fragments are closest.

4.4 Proximity neighbor selection

Section 4.2 discussed how to reduce lookup latency by using recursive lookup to avoid extra return trips. In this section we apply the observation that maintaining a larger set of possible destination nodes at each lookup hop and choosing a nearby node can reduce the latency of each hop. We'll do this by relaxing the constraints on finger table entries to create a large set of candidate nodes and choosing the nearest eligible node to fill each finger table entry, a technique often called proximity neighbor selection (PNS) and used by many DHTs [16, 50, 80, 122, 139].

It is possible to use PNS because there are usually few constraints in the choice of routing entries: any node in the relevant portion of the identifier space is eligible. A DHT design must include an algorithm to search for nearby nodes; an exhaustive search may improve lookup latency, but also consume network resources. This subsection builds on the work of Gummadi et al. [50] in two ways: it explains why PNS achieves the lookup latency that it does (approximately 1.5 times the average round trip time in the underlying network),

considers how the size of the set of candidate nodes for each entry affects latency, and shows that when using PNS, lookup latency is not strongly affected by the number of DHT nodes (and thus the number of hops in a lookup).

Early versions of Chord attempted to improve lookup latency by using un-modified routing tables but choosing low-latency routes by, for example, favoring low-latency hops that do not make as much progress in ID space (proximity route selection, or PRS, in the nomenclature of [50]). This approach did not significantly improve lookup stretch; others have confirmed our experience that PRS is less effective than PNS [50]. We will not discuss PRS further here.

Following Gummadi et al. [50], define PNS(x) as follows. The ith Chord finger table entry of the node with ID a properly refers to the first node in the ID-space range $a + 2^i$ to $a + 2^{i+1} - 1$. The PNS(x) algorithm considers up to the first x nodes in that range (there may be fewer than x), and routes lookups through the node with lowest latency. Ideal PNS refers to PNS(x) with x equal to the total number of nodes, so that every finger table entry points to the lowest-latency node in the entire allowed ID-space range. The simulator simply chooses the lowest-latency of the x nodes, while the real implementation asks each proper finger entry for its successor list and uses Vivaldi to select the closest node. This means that the real implementation requires that $x \leq$ the number of successors.

4.4.1 Ideal PNS

Figure 4-7 shows the simulated effect of PNS(x) on lookup latency for a variety of x values. For each x value, 20,000 lookups were issued by randomly selected hosts for random keys. Each lookup is recursive, goes to the key's predecessor node (but not successor), and then directly back to the originator. The graph plots the median, 10th percentile, and 90th percentile of latency.

Figure 4-7 shows that PNS(1) has a simulated average latency of 489 ms, PNS(16) has an average latency of 224 ms, and PNS(2048) has an average latency of 201 ms. The latter is ideal PNS, since the neighbor choice is over all nodes in the simulation. We'll focus first on ideal PNS.

Why does ideal PNS show the particular improvement that it does? Using our understanding of how the number of choices affects latency we can begin to build up the expected latency of a lookup. The return trip from the predecessor to the originator has the same median as the one-way delay distribution of the nodes in the network, δ:

$$\delta + \ldots$$

For the King data set, $\delta = 67$ms. The last hop (to the predecessor) has only one candidate, so its median latency is also δ:

$$\delta + \delta + \ldots$$

Each preceding hop has twice as many candidate nodes to choose from on average, since the finger-table interval involved is twice as large in ID space. So the second-to-last hop is the smaller of two randomly chosen latencies, the third-to-last is the smallest of four, etc. As we showed in Section 4.3, the minimum of n samples has its median at approximately the $\frac{1}{2n}$ percentile of the original distribution for large n. Doubling the sample size x will halve the percentile of the best sample. Assuming a uniform latency distribution, doubling the sample size halves the best sampled latency. Therefore, the latencies incurred at successive lookup

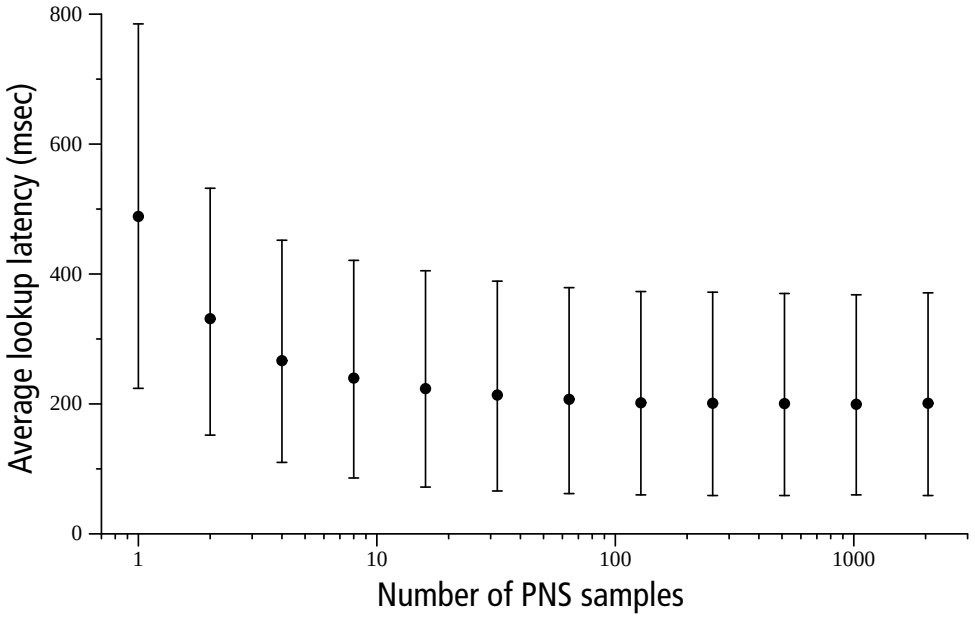


Figure 4-7: Average lookup latency as a function of the number of PNS samples. The bar at each x value shows the 10th, average, and 90th percentile of the latencies observed by 20,000 recursive lookups of random keys from random nodes using PNS(x). The measurements are from the simulator with 2048 nodes.

hops with ideal PNS can be approximated by a geometric series with the final lookup hop to the key's predecessor being the longest hop. The lookup process includes an additional final hop to the originator. If we use the per-hop median latency as a gross approximation of the average per-hop latency, the total average lookup latency is thus approximated as:

$$\delta + (\delta + \frac{\delta}{2} + \frac{\delta}{4} + ...) = \delta + 2\delta = 3\delta$$

For the King data set, this gives 201 ms. This number is coincidentally the same as our simulation results for PNS(2048).

Figure 4-8 plots the Vivaldi coordinates for the nodes participating in the DHash implementation running on PlanetLab. One finger is shown for each node (the one that points half-way around the identifier space) as a solid arrow. The dotted lines show the successor pointer of each node. Since fingers are chosen to point to nearby nodes we expect that there will not be many fingers that point to nodes far away in coordinate space. Compared to successor pointers that are not chosen based on proximity, fingers are more likely to be nearby in coordinate space: note that few finger pointers of nodes in the large cluster at the lower left point out of the cluster while many successor pointers do point to nodes far outside the cluster.

4.4.2 Sample size

From Figure 4-7 we see than the latency of PNS(16) comes relatively close to the latency given by ideal PNS, and is convenient to implement in the real system with successor lists. Why are there diminishing returns in Figure 4-7 beyond roughly PNS(16)?

One reason is that the last few hops of a lookup dominate the total latency and PNS is unable to effectively optimize these hops. The last two hops account for two-thirds of the

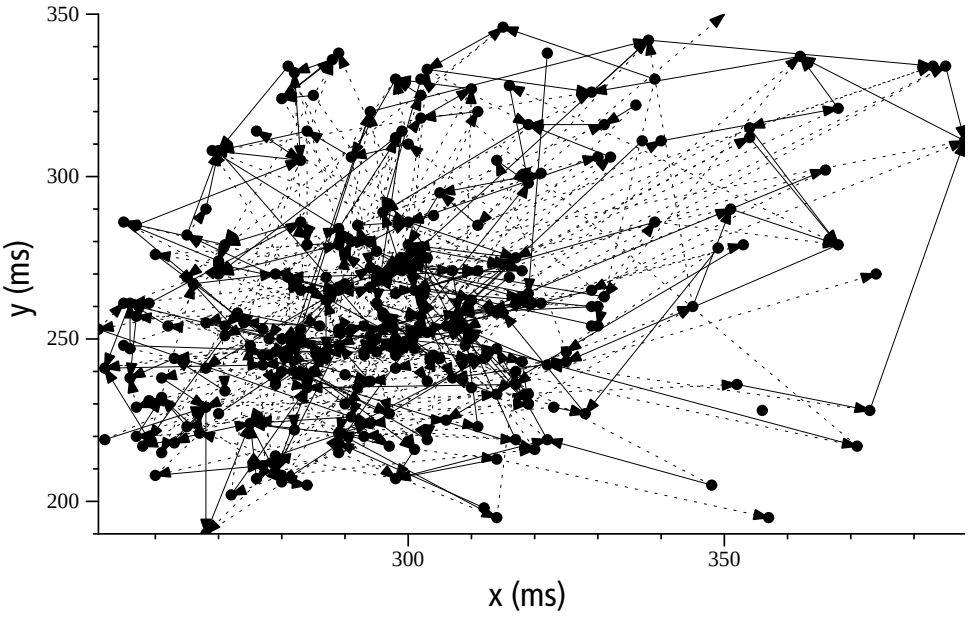


Figure 4-8: The fingers (solid lines) and successors (dotted lines) chosen by nodes running on the PlanetLab testbed. The node positions are shown in a 2-d projection of Vivaldi coordinate space from the height model. Because of PNS, a node's fingers are more likely to point to a nearby node than a successor. Only one finger per node is shown, it is the one that points half-way around the identifier space.

total latency of a lookup and PNS cannot optimize these hops at all. As a lookup gets close to the target key in ID space, the number of remaining nodes that are closer in ID space to the key, and thus eligible next hops, decreases. For large x, during the later hops of a lookup the number of samples is often limited by the number of eligible nodes, rather than by x. This limits the ability of PNS to take advantage of a large number of samples. Section 4.6 shows how to increase the size of the set of eligible nodes for the final hops.

Another reason is the small absolute reduction in per-hop latency as more samples are added coupled with a small number of hops. The median hop latency when sampling 32 nodes is about 9ms less than when sampling 16 (we can see this in Figure 4-6 by comparing where the lines labeled 16 and 32 meet the x axis). In addition, with 2000 nodes we expect about 5 hops ($0.5 * \log 2000$), of which only 2 or 3 will be able to sample at least 32 nodes due to ID-space constraints. Thus we might expect moving from 16 to 32 samples to reduce latency by less than 20ms. Compared to the $2 \times \delta$ ms that the lookup must spend to hop to the predecessor and then to the requesting node this is a small improvement. The simulation results presented in Figure 4-7 show that the actual improvement in sampling 32 nodes instead of 16 were similar to what this rough analysis predicts: 10ms.

4.4.3 Effect of scale

The fact that the average lookup latency of PNS(N) can be approximated as an infinite geometric series whose sum converges quickly suggests that despite the fact that the number of lookup hops scales as $\log(N)$, the total average lookup latency will stay close to 3δ. Figure 4-9 shows the simulated average lookup latency as a function of the number of nodes in the system. As we can see, there is indeed little increase in average lookup latency as the network grows.

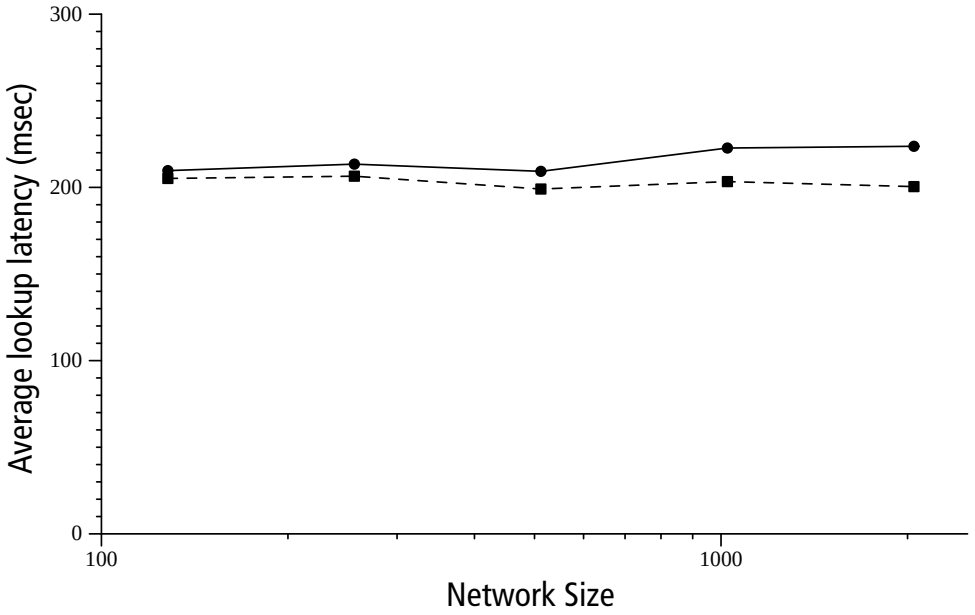


Figure 4-9: Average lookup latency of PNS(16) and PNS(N) as a function of the number of nodes in the system, N. The simulated network sizes consist of 128, 256, 512, 1024, 2048 nodes.

The difference in lookup latency using PNS(16) and PNS(N) grows slowly as the number of nodes increases because there is a small gain in median per-hop latency to be had by sampling more nodes (as discussed above). As the number of nodes grows the hop-count grows as well: the additional per-hop cost incurred by PNS(16) causes the lookup cost to grows proportionally.

The fact that lookup latency remains relatively constant with an increasing number of nodes in the system is important to consider when comparing $\log N$ hop DHTs against constant hop variants [53, 54]. By maintaining more state and reducing hop counts, these DHTs can achieve lower lookup latencies than their $\log N$ counterparts (as low as 2δ). Recall that we defined a lookup as the operation of sending a message to the node responsible for a key; if a lookup operation were only determining which node should store the data, a one-hop scheme could perform the operation with no network delay. The latency reduction provided by one-hop DHTs comes at the cost of maintaining information about a larger number of nodes. While this thesis does not consider this tradeoff in detail, it is interesting to note that the latency penalty (relative to one-hop systems) for using a multi-hop DHT is a constant δ regardless of the number of nodes in the system while the cost of maintaining state about N nodes is likely to grow with N.

4.5 Coding versus replication

Once the node originating a fetch acquires the key's predecessor's successor list, it knows which nodes hold the block's replicas [34, 112] or fragments of an erasure-coded block [21, 6, 69, 55]. In the case of replication, the originator's strategy should be to fetch the required data from the successor with lowest latency. The originator has more options in the case of coded fragments, but a reasonable approach is to fetch the minimum required number of fragments from the closest successors. An alternative would be to request more fragments

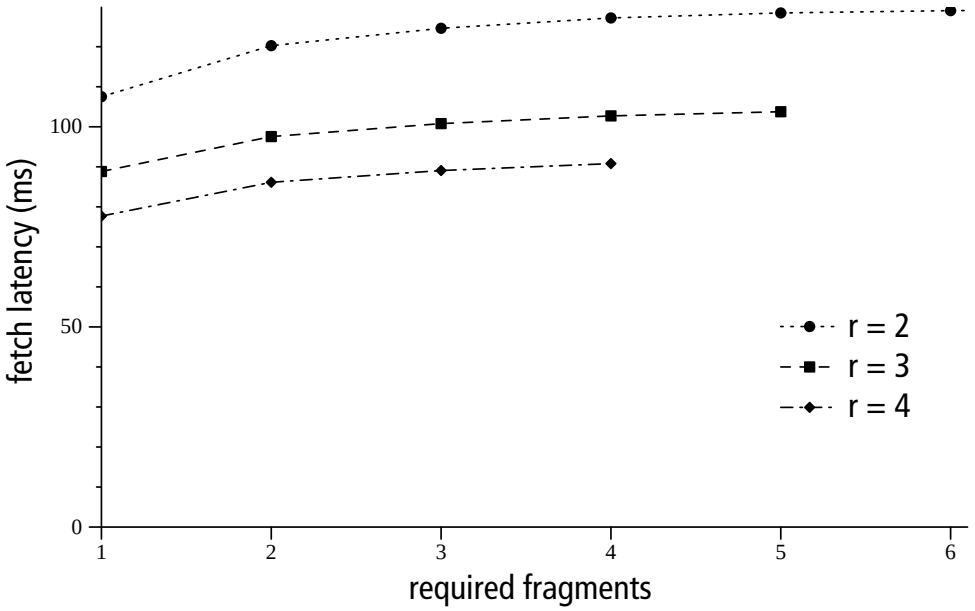


Figure 4-10: The relationship between fetch latency and l,m,r. As the number of fragments that are required to reconstruct increases, the expected fetch latency also increases. As more fragments are fetched, the probability that one of them will come from a slow server increases.

than are necessary to reconstruct the block and use the first fragments to arrive; this alternative is more reasonable when coding is used since the requested, but unused, fragments are smaller than whole replicas. The technique of fetching the data from the nearest of a set of candidate nodes is often called server selection.

The design choice here can be framed as choosing the coding parameters l and m, where l is the total number of fragments stored on successors and m is the number required to reconstruct the block. Replication is the special case in which $m = 1$, and l is the number of replicas. The *rate* of coding, $r = \frac{l}{m}$, expresses the amount of redundancy. A replication scheme with three replicas has $m = 1$, $l = 3$, and $r = 3$, while a 7-out-of-14 IDA coding scheme has $m = 7$, $l = 14$, and $r = 2$.

The choice of parameters m and l has three main effects. First, it determines a block's availability when nodes fail [135]; we'll explore this tradeoff in Chapter 6. Second, increasing r is likely to decrease fetch latency, since that provides the originator more choices from which to pick a nearby node. Third, increasing r increases the amount of communication required to write a block to the DHT.

Figure 4-10 illustrates the relationship between total fetch latency, the amount of redundancy, and the rate of encoding. Three curves are shown, corresponding to redundancy levels of 2, 3, and 4. Each point corresponds to a different way to achieve that redundancy: the x-axis value of the point is the number of fragments required to reconstruct the block (m). Thus the left-most point corresponds to replication. Points further to the right correspond to a scheme that creates $m \times r$ fragments. Each point's y-axis value is the average latency from 20,000 simulated random block fetches. The originator performs a lookup to obtain the list of the desired key's successors, then issues parallel RPCs to the m of those successors that have lowest latency, and waits for the last of the RPCs to complete. The y-axis values include only the data fetch time.

```
a.lookup(q, k, d):
  overlap = {n' | n' ∈ succlist_a ∧ n' > k}
  if |overlap| ≥ d then
    return overlap to the originator q
  else if overlap ≠ ∅ then
    t = {the s − d nodes in succlist_a immediately preceding k} ∪ overlap
    b = t_i ∈ t s.t. dist(a, t_i) is minimized
    if b ∈ overlap then
      t = b.get_succlist()
      u = merger of t and overlap to produce k first d successors
      return u to the originator q
    else
      return b.lookup(q, k, d)
  else
    b = closestpred(lookupfinger, k)
    return b.lookup(q, k, d)
```

Figure 4-11: Recursive lookup that returns at least d fragments of key k to sender q. Each node's successor list contains s nodes.

Additional redundancy (greater r) and a smaller number of required fragments (smaller m) give the fetching node more choice and result in lower latency. The best trade-off between replication and coding is dependent on the workload: a read-intensive workload will experience lower latency with replication, while a write-intensive workload will consume less network bandwidth with coding. DHash uses IDA coding with $m = 7$ and $l = 14$. The number seven is selected so that a fragment for an 8 KB block will fit in a single 1500-byte packet, which is important for UDP-based transport. The originator uses Vivaldi (Section 3) to predict the latency to the successors.

4.6 Integrating lookup and fetching

So far the design of the DHT lookup algorithm and the design of the final data server-selection have been considered separately. One problem with this approach is that obtaining the complete list of a key's s successors requires that the originator contact the key's predecessor, which Section 4.4 observed was expensive because the final lookup steps can take little advantage of proximity routing. However, of the s successors, only the first l immediate successors store the fragments for the key's data block. Furthermore, fragments from any m of these successors are sufficient to reconstruct the block. Each of the $s-m$ predecessor nodes of the key has a successor list that contains m successors. Thus the lookup could stop early at any of those predecessors, avoiding the expensive hop to the predecessor; Pastry/PAST uses a similar technique [112].

However, this design choice decreases the lookup time at the expense of data fetch latency, since it decreases the number of successors (and thus fragments) that the originator can choose from. Once the recursive lookup has reached a node n_1 whose successor list overlaps the key, n_1 is close enough to be the penultimate hop in the routing. By forwarding the query to the closest node n_2 in its successor list that can return enough nodes, n_1 can ensure that the next hop will be the last hop. There are two cases — if n_2 is past the key, then n_1 must directly

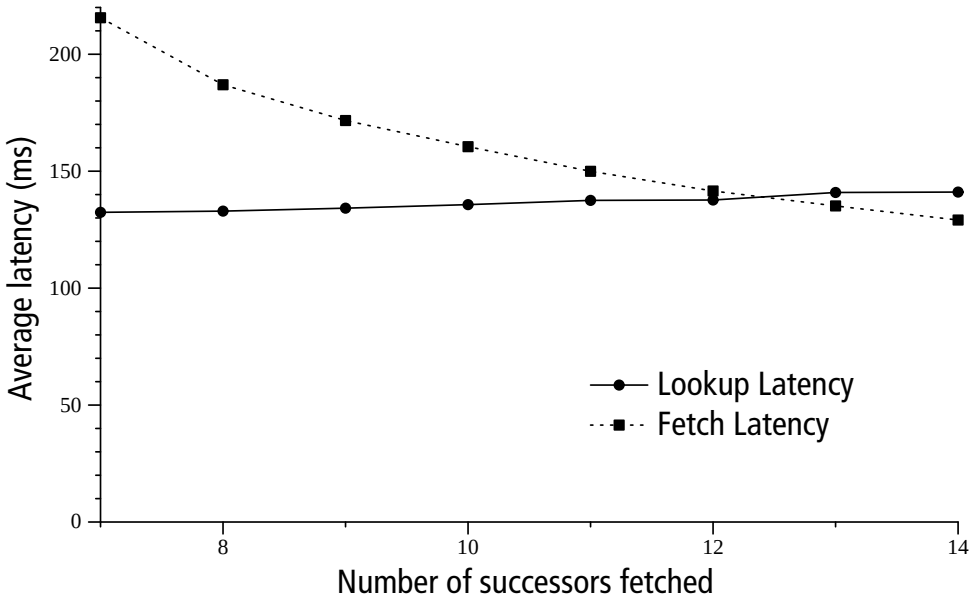


Figure 4-12: Simulated lookup and fetch time as a function of the d parameter in Figure 4-11. Larger d causes the lookup to take more hops and gather more successors; the extra successors decrease the fetch latency by providing more choice of nodes to fetch from. For comparison, the average lookup and fetch times that result from always contacting the predecessor are 224 and 129 milliseconds, respectively.

retrieve n_2's successor list and merge it with its own overlapping nodes to avoid routing a lookup beyond the key in identifier space. Otherwise, n_1 can simply hand-off the query to n_2 who will have enough information to complete the request.

Figure 4-11 shows the pseudo-code for this final version of the DHash lookup algorithm. The d argument indicates how many successors the caller would like. d must be at least as large as m, while setting d to l retrieves the locations of all fragments.

The final latency design decision is the choice of d. A large value forces the lookup to take more hops, but yields more choice for the data fetch and thus lower fetch latency; while a small d lets the lookup finish sooner but yields higher fetch latency. Figure 4-12 explores this tradeoff. The figure shows that the cost of a higher d is low: the lookup algorithm in Figure 4-11 uses only nearby nodes as the final hops, while the decrease in fetch time obtained by using larger d is relatively large. Thus setting $d = l$ is the best policy.

4.7 Summary

Figure 4-13 summarizes the cumulative effect of the design decisions explored in this section. The leftmost bar in each triple shows the median time on our PlanetLab implementation (copied from Figure 1-4). The middle bar was produced by the simulator using a latency matrix measured between PlanetLab hosts. The dark portion of each bar shows the lookup time, and the light portion shows the time taken to fetch the data. Although the simulator results do not match the PlanetLab results exactly, the trends are the same. The results differ because the simulator uses inter-host delays measured between a slightly different set of PlanetLab nodes than were used for the implementation experiments, and at a different time. In addition, the implementation of integration is less efficient that the simulated version and send an additional RPC that the simulated version does not.

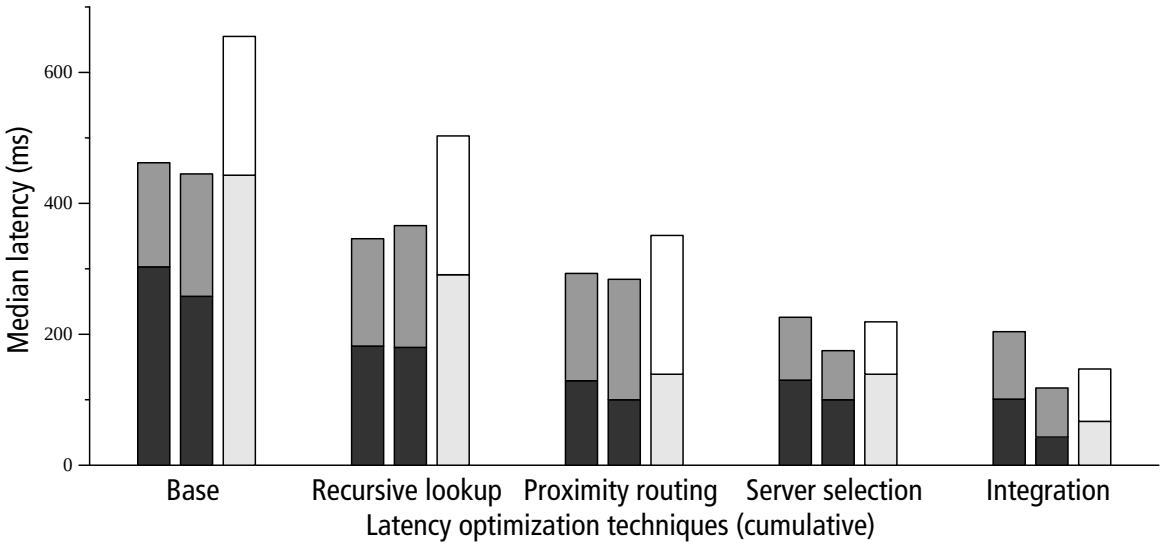


Figure 4-13: The cumulative effect of successive performance optimizations on the median latency of a DHash data fetch. The leftmost bar in each triple shows the time on our PlanetLab implementation (copied from Figure 1-4). The middle bar was produced by the simulator using a latency matrix measured between PlanetLab hosts. The rightmost bar corresponds to simulations of 2048 nodes using the King latency matrix. The dark portion of each bar shows the lookup time, and the light portion shows the time taken to fetch the data.

The rightmost bar corresponds to simulations of 2048 nodes using the King latency matrix. The absolute numbers are larger than for the PlanetLab results, and perhaps more representative of the Internet as a whole, because the King data set includes a larger and more diverse set of nodes. Again, the overall trends are the same.

The result of these optimization is a factor of two decrease in `get` latency. This delay is important since it is the waiting time a user will experience before UsenetDHT can display an article. Intuitively, shorter delays are more desirable for users. Studies of user satisfaction with an online learning system show that users of a system with a 250ms latency were more satisfied than users of a system with 1.3s latencies [136].

5

Throughput

Some of the applications using DHash store a large amount of data that may be downloaded in bulk (large binary Usenet postings, for instance). This data is striped across many machines, and traditional stream-oriented network protocols proved not to be suitable for downloading data spread across a large number of servers. DHash, when using these protocols, provided low throughput and caused high loss on the network. This chapter discusses why these protocols performed poorly and presents an alternative protocol, the striped transport protocol (STP), designed explicitly for DHTs.

5.1 Problem

Stream-oriented network protocols do not work well with DHash because DHash makes many short-lived connections to a large number of nodes. Dhash's design encourages applications to communicate with many hosts: large objects are likely to be stored in DHash as a collection of small blocks since DHash provides the best load-balance with small blocks. A multi-megabyte file in CFS will be broken into thousands of 8K blocks. DHash's load balancing properties naturally spread these blocks over a large number of machines and further encode them into fragments; downloading the file requires contacting thousands of hosts and receiving a small (MTU-sized) packet from each. Figure 5-1 shows schematically the flow of information during a striped download: many hosts send data to one host.

This communication pattern causes stream-oriented transports to behave poorly; in particular, they achieve low-throughput and caused high packet loss. TCP, for example, is designed for long-term communication between a pair of hosts. The design of TCP reflects this goal: TCP attempts to estimate the round-trip time by measuring several round-trips, for instance. If an application exchanges only one or two packets on a connection, TCP will never find a good estimate for the round-trip time. TCP's estimate of the available bandwidth behaves similarly; the first several packets of a stream are exchanged in slow-start and are not subject to flow or congestion control. These protocols also reserve resources on a per-connection basis; this design makes sense under the assumption that connections will be long-lived.

The initial DHash implementation used TCP as a transport by opening (and potentially caching) a connection to each contacted node. As the number of nodes in the system grew, this strategy quickly led to resource exhaustion. DHash was forced to contact, and make a TCP connection to, a large number of machines in a short period of time. The system quickly exhausted the available file descriptors.

Even if these resource shortages are avoided (by aborting TCP connections, for instance) using TCP, or any number of stream-oriented protocols [91, 68], has two drawbacks. First,

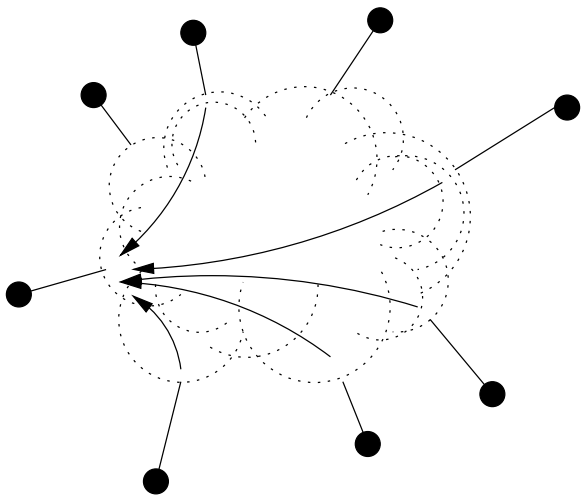


Figure 5-1: The download of a large data object in DHash involves a single node communicating with many other nodes. Each communication is short and, in a large system, a node is not likely to communicate with the same node twice within a short time frame.

TCP has a high connection setup cost. Also, when TCP is used to transfer only a very small amount of data the various estimators used by the transport (RTT, available bandwidth) do not have time to converge to good estimates. In particular, TCP connections remain in slow-start and the congestion avoidance properties of the transport are not effective. Keeping a large number of TCP connections in this state led to high packet loss due to congestion. Essentially, the very short-lived connections were not performing any type of congestion avoidance. Second, keeping a large number of connections open (even long-running connections) leads to unfair bottleneck link sharing. The most promising approach to using TCP is to maintain a small number of long-lived connections to hosts present in a node's routing table. This approach requires data to be moved in and out of hosts' access links leading to high latency and low throughput; we'll evaluate this approach in detail in Section 5.3.

A commonly used alternative to stream-oriented protocols is a datagram protocol such as UDP. Using UDP as a transport eliminates the resource-utilization problems: no per-host state is maintained and only a single file descriptor is used. However, a UDP transport is equally prone to congest links leading to high loss rates and low throughput.

5.2 STP

A good transport for the DHT must satisfy requirements are similar to those of traditional unicast transport protocols such as TCP [61], but with the additional requirement that the solution function well when the data is spread over a large set of servers. The transport should keep enough data in flight to cover the network's delay-bandwidth product, stripe data over multiple slow access links in parallel, and recover in a timely fashion from packet loss. The transport must also provide congestion control in order to avoid unnecessary re-transmissions and to avoid overflowing queues and forcing packet loss.

DHash uses a custom transport (STP) that meets these requirements. STP allows nodes to put and get data directly to other nodes, rather than routing the data through multiple overlay hops. STP does not maintain any per-destination state; instead, all of its decisions

are based on aggregate measurements of recent network behavior, and on Vivaldi latency predictions. STP's core mechanism is a TCP-like congestion window controlling the number of concurrent outstanding RPCs.

While STP borrows many ideas from TCP, DHT data transfers differ in important ways from the unicast transfers that TCP is designed for. Fetching a large quantity of DHT data involves sending lookup and get requests to many different nodes, and receiving data fragments from many nodes. There is no steady "ACK clock" to pace new data, since each RPC has a different destination. The best congestion window size (the number of outstanding RPCs to maintain) is hard to define, because there may be no single delay and thus no single bandwidth-delay product. Quick recovery from lost packets via fast retransmit [120] may not be possible because RPC replies are not likely to arrive in order. Finally, averaging RPC round-trip times to generate time-out intervals may not work well because each RPC has a different destination.

5.2.1 STP window control

Each DHash server controls all of its network activity with a single instance of STP. STP maintains a window of outstanding UDP RPCs: it only issues a new RPC when an outstanding RPC has completed. STP counts both DHT lookup and data movement RPCs in the window.

STP maintains a current window size w in a manner similar to that of TCP [61, 23]. When STP receives an RPC reply, it increases w by $1/w$; when an RPC must be retransmitted, STP halves w. Applications do not need to be aware of the current value of w but, if they require high throughput, should issue at least w RPCs in parallel to STP. In practice, applications maintain a large, fixed window of RPCs (usually around 32). Applications could issue an unbounded number of RPCs in parallel to STP without damaging the network, but would see large delays as STP must wait for room in the window to issue the RPCs.

STP actually keeps $3w$ RPCs in flight, rather than w. Using w would cause STP to transfer data significantly slower than a single TCP connection: lookup RPCs carry less data than a typical TCP packet, STP has nothing comparable to TCP's cumulative acknowledgments to mask lost replies, STP's retransmit timers are more conservative than TCP's, and STP has no mechanism analogous to TCP's fast retransmit. The value 3 was chosen empirically to cause STP's network use to match TCP's. It might be possible to avoid artificially increasing the window size by only placing data RPCs under STP's congestion control. Because data RPCs are likely to be MTU-sized the multiplier would not be necessary. Some lookup systems enforce bandwidth usage limits on lookup traffic with their own mechanisms (Accordion [72], notably); these systems are unlikely to congest links with lookup traffic if the bandwidth budget is set low enough.

The size of a TCP congestion window lends itself to straightforward interpretation: it is the protocol's estimate of the available bandwidth between the source and destination. STP does not operate with a stable source-destination pair, so such an interpretation does not make sense when discussing STP.

Under certain assumptions, however, the STP window size can be understood intuitively. For instance, when a single client is downloading a file in an otherwise idle system, that client's access link is likely to be the bottleneck for the entire striped transfer. In this case, the window size corresponds to the available bandwidth on the access link. STP was designed with this assumption in mind. When it is true, STP fully utilizes a client's access link. This assumption is often, but not always true, however. For instance, in a small system (like the RON test bed evaluated here), a remote node could be the bottleneck for a single striped

download if the client is able to download blocks from each remote node fast enough to saturate the access link of the slowest remote node. This implies that the client's access link is N times greater than the slowest remote node (since, on average, the client only contacts the slowest node to fetch every Nth block). In this case, STP's window corresponds to N times the available capacity on the remote node. In this scenario, the client's own access link is underutilized.

5.2.2 Retransmit timers

Lost packets have a negative impact on DHash throughput. STP can not perform fast retransmits and every loss therefore causes STP to return to slow start. Also, because each block transfer is preceded by a multi-RPC lookup even a modest packet loss rate has an impact on throughput since a successful block download requires that at least $(O(\log N))$ packet transmissions succeed. Because STP is sensitive to lost packets, setting retransmit timers accurately is important to obtaining high throughput. Ideally STP would choose timeout intervals slightly larger than the true round trip time, in order to to waste the minimum amount of time. This approach would require a good RTT predictor. TCP predicts the RTT using long-term measurements of the average and standard deviation of per-packet RTT [61]. STP, in contrast, cannot count on sending repeated RPCs to the same destination to help it characterize the round-trip time. In order for STP to perform well in a large DHT, it must be able to predict the RTT before it sends even one packet to a given destination.

STP uses Vivaldi latency predictions to help it choose the retransmit time-out interval for each RPC. However, Vivaldi tends to under-predict network delays because it does not immediately account for current network queuing delays or CPU processing time at each end. Since under-predicting the latency of an RPC is costly (a spurious loss detection causes a halving of the current window) STP adjusts the Vivaldi prediction before using it. STP characterizes the errors that Vivaldi makes by keeping a moving average of the difference between each successful RPC's round-trip time and the Vivaldi prediction. STP keeps this average over all RPCs, not per-destination. STP chooses an RPC's retransmission interval in milliseconds as follows:

$$RTO = \nu + 6 \times \alpha + 15$$

where ν is the Vivaldi-predicted round trip time to the destination and α is the average error. The weight on the α term was chosen by analyzing the distribution of RPC delays seen by a running node; the chosen timers produce less than 1 percent spurious retransmissions with approximately three times less over-prediction in the case of a loss than a conservative (1 second) timer. This formula assumes that Vivaldi's errors are normally distributed; adding a constant times the error corresponds to sampling a low percentile of the error distribution. The constant α plays a part similar to the measured RTT deviation in the TCP retransmit timer calculation.

The constant term in the calculation of RTO (15 ms) is necessary to avoid retransmissions to other virtual nodes on the same host; Vivaldi predicts small latencies to the local node, but under high load the observed delay is as much as 15 ms mainly due to the large latency of database operations to store and retrieve blocks. This term prevents those retransmissions without adding significantly to over-prediction for distant nodes.

5.2.3 Retransmit policy

When an STP retransmit timer expires, STP notifies the application (DHash) rather than re-sending the RPC. This design gives DHash a chance to re-send the RPC to a different destination. DHash re-sends a lookup RPC to the finger that is next-closest in ID space, and re-sends a fragment fetch RPC to the successor that is next-closest in predicted latency. This policy helps to avoid wasting time sending RPCs to nodes that have failed or have overloaded access links.

DHash uses a separate background stabilization process to decide whether nodes in the finger table or successor list have crashed; it sends periodic probe RPCs and decides a node is down only when it fails to respond to many probes in a row.

5.2.4 Selective use of TCP

In practice, applications occasionally store very large blocks (greater than 100KBytes); Usenet news articles posted to binary newsgroups range in size up to 1MB and the implementation of UsenetDHT stores these articles as a single block. Storing such blocks requires sending a large number of packets to the same host: the default erasure coding parameters used by DHash result in replicas that are 1/7th the size of the original block.

Sending the block as a single IP packet is not feasible: the maximum IP packet size is smaller than some replicas, and, additionally, allowing IP to fragment the block into MTU-sized chunks results in a reduced delivery probability (if one fragment is lost, the entire block must be resent).

One way to store these large blocks would be to send multiple, MTU-sized RPCs. STP will dispatch these RPCs at the appropriate rate. However, in this scenario, per-host information (that STP is designed not to store) is readily available. STP allows applications that know that they will send a large number of packets to the same host to specify that TCP should be used. DHash currently requests a TCP connection when storing or fetching a block that will result in a fragment greater than 8 KBytes.

5.3 Performance comparison

This section presents measurements comparing the latency and throughput of the TCP transport implementation to the STP implementation when run on the RON test-bed [4]. We used 26 RON nodes, located in the United States and Europe. Each physical RON node is located in a different machine room and ran 4 copies of DHash. The average inter-node round-trip time is 75 ms, and the median is 72 ms (these reflect the multiple copies of DHash per host).

5.3.1 Persistent TCP

One way to avoid the problems with TCP would be to maintain persistent connections to a node's finger table and restrict all communication to those links. Restricting communication to the overlay links means that all lookups and data movement must be recursive: iterative lookups or direct movement of data would not be able to use the persistent inter-neighbor TCP connections. Recursive lookups work well to reduce lookup latency. However, recursive data movement requires that each block of data be returned through the overlay rather than directly. This recursive return of data causes it to be sent into and out of each hop's Internet

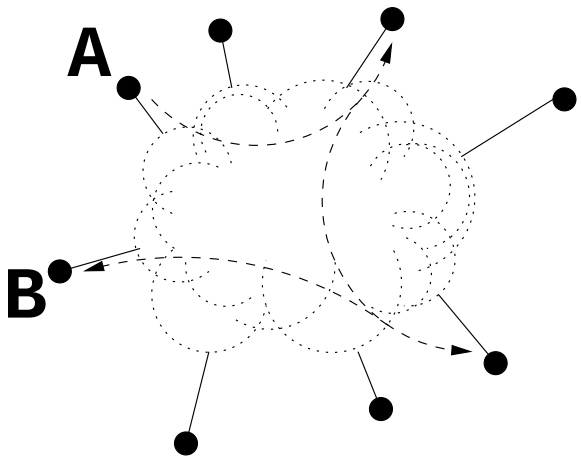


Figure 5-2: The path of a block stored at A and downloaded by B. Note that the block crosses the underlying network several times and also traverses intermediate access links

access link, potentially increasing latency and decreasing useful throughput. Figure 5-2 shows that path that a block takes when returning to the originator using recursive data movement.

DHash allows the option to use TCP as the transport. Each node keeps a TCP connection open to each of its fingers, as well as a connection to each node in its successor list. DHash forwards a get request recursively through neighbors' TCP connections until the request reaches a node whose successor list includes a sufficient number of fragments (as in Section 4.6). That node fetches fragments in parallel over the connections to its successors, trying the most proximate successors first. It then re-constructs the block from the fragments and sends the block back through the reverse of the route that the request followed. Pond [107] moves data through the Tapestry overlay in this way.

5.3.2 Fetch latency

Figure 5-3 shows the distribution of individual block fetch latencies on RON. The numbers are derived from an experiment in which each node in turn fetched a sequence of randomly chosen blocks; at any given time only one fetch was active in the DHT. The median fetch time was 192 ms with STP and 447 ms with TCP. The average number of hops required to complete a lookup was 3.

The STP latency consists of approximately 3 one-way latencies to take the lookup to the predecessor, plus one one-way latency to return the lookup reply to the originator. The parallel fetch of the closest seven fragments is limited by the latency to the farthest fragment, which has median latency (see Section 4.5). Thus the total expected time is roughly $4 \times 37.5 + 72 = 222$; the actual median latency of 192 ms is probably less because of proximity routing of the lookup.

The TCP latency consists of the same three one-way latencies to reach the predecessor, then a median round-trip-time for the predecessor to fetch the closest seven fragments, then the time required to send the 8 KB block over three TCP connections in turn. If the connection uses slow-start, the transfer takes 2.5 round trip times (there's no need to wait for the last ACK); if not, just half a round-trip time. A connection uses slow-start only if it has been idle for a second or more. The connection from the first hop back to the originator is typically not idle, because it has usually been used by a recent fetch in the experiment; the other

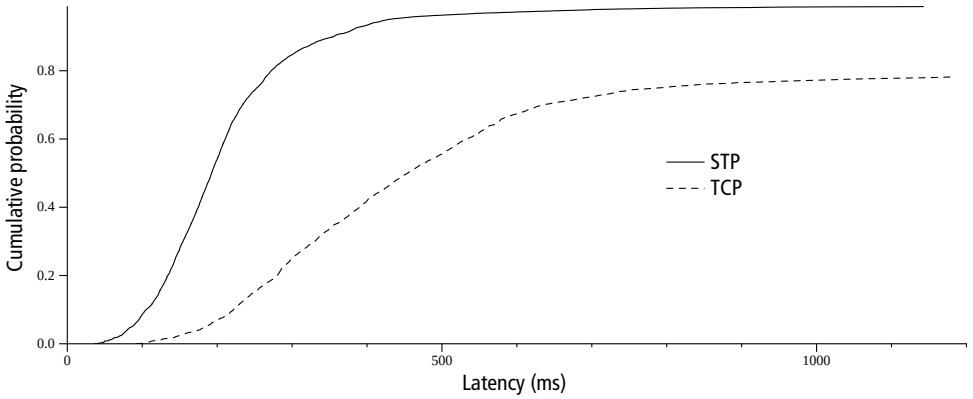


Figure 5-3: Distribution of individual 8192-byte fetch latencies on RON.

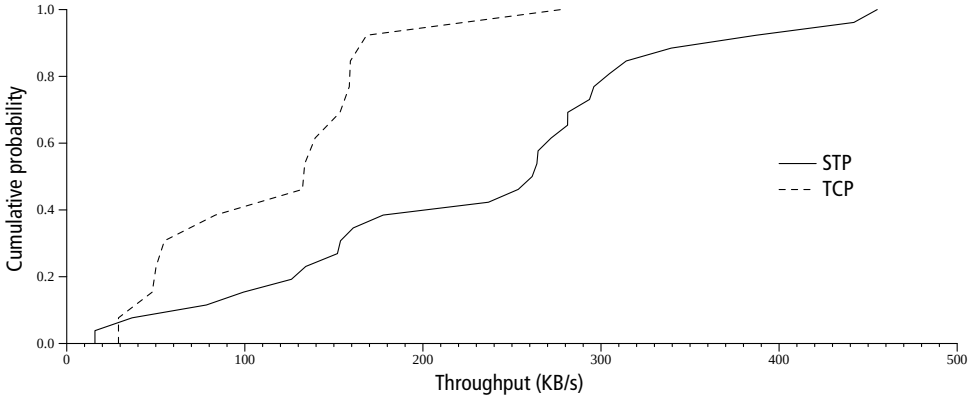


Figure 5-4: Distribution of average throughput obtained by different RON nodes during 4 megabyte transfers.

connections are much more likely to use slow start. Thus the latency should range from 340 ms if there was no slow-start, to 600 ms if two of the hops used slow-start. The measured time of 447 ms falls in this range. This analysis neglects the transmission time of an 8 KB block (about 131 ms at 1 Mb/s).

5.3.3 Single-client fetch throughput

Figure 5-4 shows the distribution of fetch throughput achieved by different RON nodes when each fetches a long sequence of blocks from DHash. The application maintains 64 one-block requests outstanding to its local DHash server, enough to avoid limiting the size of STP's congestion window.

Using TCP transport, the median node achieves a throughput of 133 KB/s. The minimum and maximum throughputs are 29 and 277 KB/s. Both the median throughput and the range of individual node throughputs are higher when using STP: the median was 261 KB/s, and throughputs range from 15 to 455 KB/s. The TCP transport has lower throughput because it sends each block back through each node on the recursive route, and is more likely than STP to send a block through a slow access link. About half of the three-hop routes pass through one of the RON sites with sub-one-megabit access links. STP sends fragments directly to the node originating the request, and thus each fragment encounters fewer slow links.

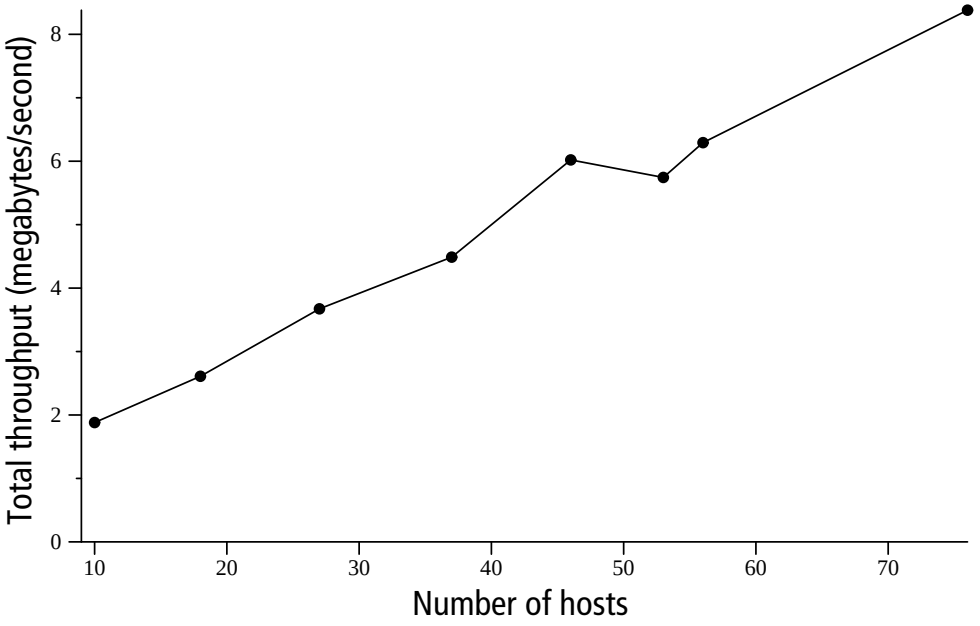


Figure 5-5: The effect of system size on total throughput obtainable. Each point represents an experiment with DHash running at x sites on the RON and PlanetLab test-beds. Each site reads 1000 8 KB blocks; the aggregate throughput of the system in steady state is reported. This throughput increases as additional capacity (in the form of additional sites) is added to the system.

To characterize the effectiveness of STP in utilizing available resources we consider the expected throughput of a DHash system. Assuming an STP window large enough to keep all links busy (and no packet loss), a node can fetch data at a rate equal to the slowest access link times the number of nodes, since the blocks are spread evenly over the nodes.

The slowest site access link in RON has a capacity of about 0.4 Mb/s. With 26 nodes one would expect $0.4 \times 26 = 10.4$ Mb/s or 1.3 MB/s total throughput for a fetching site not limited by its own access link. STP achieves less than half of this throughput at the fastest site. The reason appears to be that STP has difficulty maintaining a large window in the face of packet loss, which averages about 2 percent in these tests.

5.3.4 Scale

This section evaluates the ability of STP and DHash to take advantage of additional resources. As the number of nodes grows, more network capacity (in the form of additional access links) becomes available. Figure 5-5 shows the total throughput for an N-node DHT when all N nodes simultaneously read a large number of blocks, as a function of N. The experiments were run on the combined PlanetLab and RON test-beds. The slowest access link was that of a node in Taiwan, which was able to send at 200 KB/s to sites in the US. The observed throughput corresponds to our throughput prediction: the total throughputs scales with the number of sites. The first data point consists of ten sites experiencing an aggregate of ten times the bandwidth available at the slowest site.

A similar experiment run using 150 machines but at 70 unique sites (many PlanetLab sites are home to more than one node) produces a peak throughput of 12.8 MB/s. Adding

more machines behind the same access-link produces a higher aggregate throughput because DHash gains a proportionally greater share of that site's link bandwidth and the system's aggregate bandwidth increases. Also, each additional node effectively increases the DHash window size and, thus, the site's throughput.

6

Replica maintenance

Replication has been widely used to increase data availability in a variety of storage systems (e.g. RAID [96], Harp [75], DDS [49], Petal [70], System R duplex disks [20]). The algorithms traditionally used to create and maintain data redundancy are tailored for the environment in which these systems operate: a small number of well-connected hosts that rarely lose data or become unavailable. In this setting it is reasonable to, for example, constantly monitor all nodes in the system and to create a new replica of a data object immediately after the unavailability of one replica is detected.

DHash does not fit this model. DHTs (such as DHash) and other large-scale systems comprise thousands of machines with poor connectivity and reliability. DHTs, in particular, are likely to be deployed over the wide-area network in environments with limited inter-node link capacity. Replication algorithms designed for the machine room can be expensive when run over wide-area networks.

However, applications like UsenetDHT require efficient replica management in wide area deployments. While UsenetDHT's potential savings relative to Usenet are large, if the cost of maintaining data in a DHT is prohibitive, the potential savings of a system like UsenetDHT could be erased.

Recall from Chapter 1 that DHash's goal is data durability. A block is durable if it is stored on some media and will be available through the system at some point in the future. This is in contrast to availability which implies that the block can be retrieved through the system. This chapter describes how to manage replicas to provide good durability with minimal bandwidth requirements for services, like DHash, that are deployed on the wide-area. In particular, we target an environment that has the following properties:

1. Limited-capacity interconnect: Nodes are connected by wide-area links with limited capacity. The most expensive operation in the system is sending a new copy of a block over a wide-area link.

2. Unreliable nodes: Nodes suffer temporary failure because wide-area links fail or hosts are shut down temporarily. Permanent failures occur because disks fail or nodes leave the system permanently. Since disk lifetime is much greater than node uptime, we expect that most failures are temporary.

3. Read-only data: Mutable data blocks are rare and have weak consistency requirements. This property allows the system to take advantage of copies that have been offline for a period of time but return to the system. Were updates common, intervening updates would have made those copies unusable.

4. Large scale: Hundreds or thousands of nodes are participating in the system and storing data. We expect the system to include enough nodes that it is infeasible to maintain a

complete group membership at any node or for every node to probe every other node in the system periodically.

While the goal of DHash's maintenance algorithms is data durability, DHash cannot directly observe durability because durable replicas might be on temporarily unavailable servers. For this reason DHash's actions are driven by the number of reachable replicas. This approach causes the system to work hard to provide good availability, a useful property in its own right. Since our underlying goal is durability, however, we'll attempt to minimize the number of replicas created in response to temporary failures that affect only availability.

This chapter makes two specific contributions to replica maintenance in wide-area storage systems. First, it develops a model for durability based on the observation that any replication system must create replicas faster than they are lost. This birth-death model predicts how durably a system can store data, predicts how many replicas are required to ensure durability, and guides decisions about how to maintain the replicas and where to place them. The main parameters to this model are the rate of disk failure and the time required to create new replicas over a limited-capacity network. While some of the parameters that effect replica creation time are out of the control of system designers (link bandwidth, for instance), we show how design decisions (such as how replica sets are chosen) influence durability.

Second, this chapter presents an algorithm, Sostenuto, that maintains durability while minimizing subsequent repairs due to temporary failures. Sostenuto lazily creates replicas beyond the number needed to protect data against permanent loss. This algorithm reduces the amount of replication traffic compared to existing techniques that create a set amount of extra redundancy at the outset but "forget" about replicas that are unreachable at the time of a repair action [10].

We show that by building DHash on top of Sostenuto, it is possible to store a large amount of data efficiently in a DHT. We evaluate the cost of running Sostenuto in simulation using a trace of failures from the PlanetLab testbed and compare Sostenuto's costs to other data maintenance algorithms. From this simulation we can extrapolate the cost of storing the complete archive of Usenet data for 4 months: storing this Usenet data requires a storage overhead factor of less than 5 (including the cost of creating initial replicas) and an additional bandwidth requirement of approximately 2.5 Mb/s.

6.1 System overview

To provide high durability, the data maintenance algorithms of any storage system must create copies of data items faster than they are destroyed by failures that cause data loss. Since we are building a system whose goal is durable storage, we are primarily concerned with failures that cause permanent data loss. These failures could be caused by disks failure, operator error, or permanent departure. We'll use the term disk failure to refer to all of these cases.

6.1.1 Feasibility

Whether the system can make new copies faster than they are destroyed depends on a variety of factors including the amount of data stored per node, the node lifetime, and the nodes' link capacity. If a node can not transfer the amount of data it is storing during its lifetime, no replication policy can prevent data from being lost. Such systems are not *feasible*. The feasibility envelope defines how much data a system can store, or how little bandwidth it

can use in the best case. In practice systems operate far from the feasibility limit in order to provide good durability at the expense of reduced capacity.

Feasible systems maintain replicas to absorb bursts of simultaneous disk failures (simultaneous, here, refers to failures that are separated in time by an interval shorter than the time required to recreate the failed replicas). Increasing the number of replicas allows the system to survive a larger number of simultaneous failures; it does not allow the system to survive a higher average disk failure rate. Increasing the replication level can not make an infeasible system feasible.

To see this, consider a system with no bursts of disk failures, that is one where failures are distributed uniformly in time. In this system two replicas of a block would either be sufficient for the system to never lose data, or the system would inevitably lose blocks regardless of how many replicas are maintained. If the time between failures is larger than the amount of time required to copy a block, maintaining two replicas provides the one additional copy necessary to maintain availability during repair. If the failure inter-arrival time is smaller than the repair time, additional replicas are of no help: during each failure the node will fail to copy some of its data. Eventually, the system will "adapt" to the failure rate by discarding enough blocks so that it becomes feasible to store the remaining blocks.

Real systems do not see failures at regular intervals. Failures occur in bursts (due to correlated events, or to chance arrivals caused by independent processes). In this environment a system can be feasible but require more than two replicas: the average arrival failure rate is low enough that the system can create new replicas faster than they are lost, but the simultaneous failure of both replicas could cause data unavailability. Understanding how many replicas to maintain is an important question: too few replicas risk data loss; too many leads to high cost. In Section 6.2 we use the birth-death model of disk failure and block creation to estimate the relationship between how many replicas the system maintains and the probability of data loss.

While increasing the number of replicas can not make an infeasible system feasible, the system has some control over how fast replicas are created. It can adjust how data is placed within the system, for example, to increase the number of network connections used in parallel to reconstruct blocks. Increasing the block creation rate can make it feasible to store more data or, if the amount of stored data remains constant, make it less likely that data will be lost. Section 6.3.2 discusses techniques to increase the replica creation rate.

For systems that are feasible, we wish to find a policy that maintains block availability while minimizing the number of copies of blocks sent across wide-area links. The particular policy we will describe is built around low- and high-watermark parameters (r_L and r_H).

The system preserves at least r_L replicas at all times: when the number of readable copies falls below r_L, the system creates new copies to raise the number of copies back to r_L. The system will often end up with additional replicas beyond r_L when a node crashes, and then returns to the system, with its data intact, after a new copy of its data was created. These "extra" replicas are maintained lazily: the system tracks up to r_H replicas, but will not act to replace a failed replica unless less than r_L are available.

6.1.2 Sources of maintenance cost

The goal of Sostenuto is to minimize the cost of maintaining data. Before proceeding, we attempt to understand the sources of maintenance traffic. The first source is traffic necessary to restore replicas lost to disk failures: the magnitude of this traffic is a function of how fast disks fail and is, under our assumption that disk failures are independent events, roughly

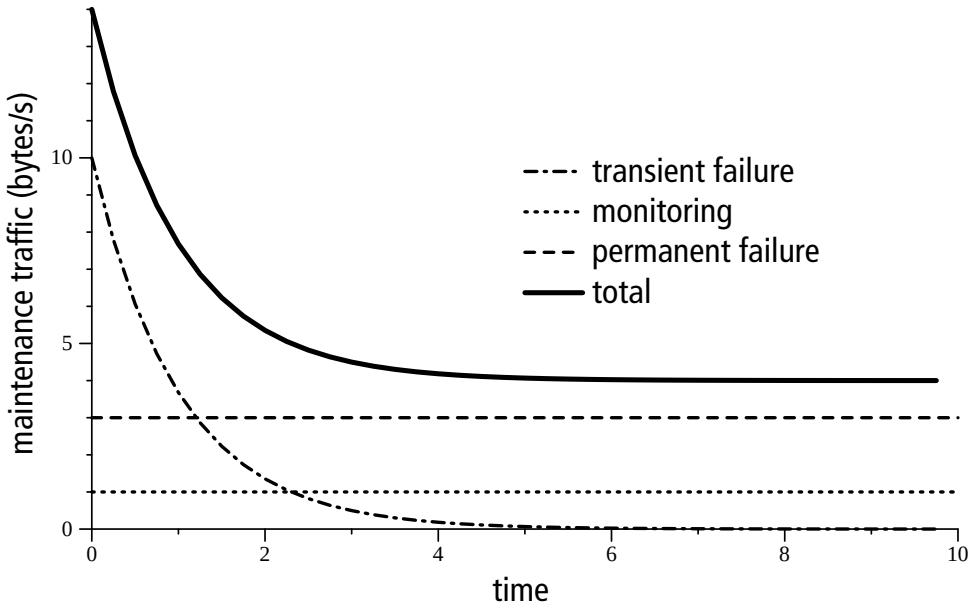


Figure 6-1: A schematic representation of the cost associated with maintaining a block over time. The x-axis of the plot is time, the y-axis indicates how much "work" the system is doing at a given moment in time; work is, under our assumptions, equivalent to network bandwidth. Maintenance cost is dominated by temporary failures early in the block's lifetime, but in the long term, disk failures make up most of the cost.

constant over time (Section 6.3 discusses how many copies to create in order preserve data despite permanent failures and how to place them).

Communication is also required to copy blocks when nodes fail temporarily. While we might expect the cost due to such failures to be proportional to the failure rate as it is for permanent failures, our lazy maintenance of blocks causes the number of such copies will drop exponentially with time (we will show this in Section 6.4). Finally, the system must send traffic on the network to monitor block availability (this cost is discussed further in Section 6.6.2).

Figure 6-1 illustrates how these costs and the total cost of maintaining a block evolve over time. Temporary failure (the dot-dash line) causes a large number of copies initially, but as more copies exist fewer and fewer copies are made. Disk failures (dashed line) and monitoring (dotted line) require a constant level of traffic. The solid line shows the total cost of maintaining a block over time: this cost is dominated by temporary failures in the short term and disk failures in the long term.

The rate of disk failures dictates a minimum replica creation rate and cost: our goal is to produce a system in which the transient and monitoring costs are small compared to the unavoidable costs due to disk failures.

6.2 r_L and the probability of data loss

In this section we define more carefully the birth-death model of block failure and repair and use it to estimate the relationship between r_L and the probability of block loss. Since the goal of our system is to preserve durability, we are primarily concerned with the disk failure rate in this model and will assume that all failures are permanent.

We will assume that disk failures occur randomly with rate λ_f and the contents of a failed

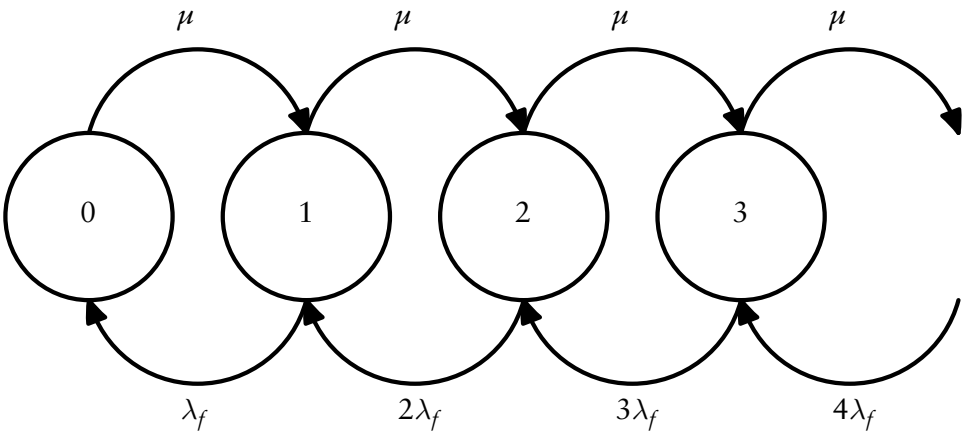


Figure 6-2: The birth-death process for replicas. Each state above corresponds to the number of replicas of a given block. Replicas are created at rate μ (governed by factors such as link capacity and the amount of data in the system). Disks fail permanently with rate λ_f. The rate of replica loss grows as more replicas are added since nodes are failing in parallel.

replica server can be recreated at rate μ. The parameter λ_f is a property of how often disks fail; μ depends on the amount of data stored at each node and the bandwidth available to create new replicas. Using these parameters the replication level of a block can be modeled as a Markov process.

6.2.1 Expected replica count

The birth-death model allows us to derive a simple property of the system: given a failure rate and creation rate, how many replicas do we expect will exist in the system? Later we'll answer a related question (which is somewhat more relevant): given r_L eagerly-maintained replicas, how likely is it that a block will be lost. Considering this simpler question first will be illustrative, however.

We'll begin our analysis of the system using a discrete-state, discrete-time, birth-death Markov chain, shown in Figure 6-2. Each state in the chain corresponds to the replication level of a single block. The system moves to higher replication levels at the rate μ. This rate is unchanged as more replicas are created since, although more replicas exist to send out data in parallel, using DHash's successor list placement scheme means that only one node can download those blocks. The rate at which transitions to lower levels are made increases as the number of replicas increases: when n replicas are present, the likelihood of any server failing increases by a factor of n since servers fail in parallel. Nodes continually create new replicas in this analysis; this corresponds to $r_L = \infty$.

Note that the transition rates are per trial, not per unit time. All of our results will be in terms of trials that have elapsed since the system started. Since there is no way to relate the number of trials to the actual time elapsed, we will not be able to answer questions such as "what is the probability of block loss after 90 days?" We consider the analysis despite this limitation because we are able to derive simple closed forms for properties that do not depend on the transient behavior of the system. We will refine this model to correspond more closely with reality later in the section.

At first glance, it might appear that if $\mu > \lambda_f$, the system will maintain an unbounded number of replicas since it can create replicas more quickly than they are destroyed. By analogy with M/M/1 queues, a replica creation corresponds to a packet arrival and the disk

failure rate to the service rate. In the queuing system, if the arrival rate is greater than the service rate, the queue length (analogous the number of replicas) grows without bound. In the birth-death model considered here, however, the expected number of available replicas is finite and is equal to μ/λ_f. The number of replicas does not grow without bound because the replica failure rate increases as the number of replicas increases.

This ratio (μ/λ_f) will be important to our discussion and will be referred to as ρ. To see why the expected number of replicas is ρ, we will derive an expression for the probability that r blocks exist and sum over all possible r. The probability that r replicas for a block exist is the probability that the system is in state r, P_r. Since this model is a birth-death process, P_r, in terms of P_0 is:

$$P_r = \frac{\mu^r}{\lambda_f^r r!} P_0.$$

The factorial comes from the coefficients on the λ_f terms. Summing rP_r (and noting that the first term of the sum‘, $r = 0$, is zero):

$$\begin{aligned} E[r] &= 0 + \sum_{r=1}^{\infty} rP_r \\ &= \sum_{r=1}^{\infty} r\frac{e^{-\rho}\rho^r}{r!} \\ &= e^{-\rho}\rho\sum_{r=1}^{\infty}\frac{\rho^{r-1}}{(r-1)!} \\ &= e^{-\rho}\cdot\rho\cdot e^{\rho} \\ &= \rho. \end{aligned}$$

To understand this result consider what ρ represents: it is the ratio between the rate at which replicas can be created and the rate at which they fail. For the system to reach a steady state we must have $\rho > 1$ or all replicas will be lost before they can be replaced. The expected number of replicas does not diverge even though the system can create replicas faster than they are lost ($\rho > 1$), because the effect of many replicas failing in parallel prevents a large number of replicas from accumulating.

This result also implies that ρ influences durability: if ρ is large, the system will operate with more replicas (farther to the right of the chain) and is less likely to experience enough simultaneous failures to eliminate all extant copies. When ρ is smaller, fewer copies are likely to exist and the system is more likely to experience a cluster of failures that cause data loss. In fact, the probability that this system is in the state (P_0) corresponding to no replicas being present decreases exponentially with ρ. Using the law of total probability,

$$\begin{aligned} 1 &= \sum_{r=0}^{\infty} P_r \\ &= P_0\sum_{r=0}^{\infty}\frac{\rho^r}{r!} \\ &= P_0 e^{\rho}. \end{aligned}$$

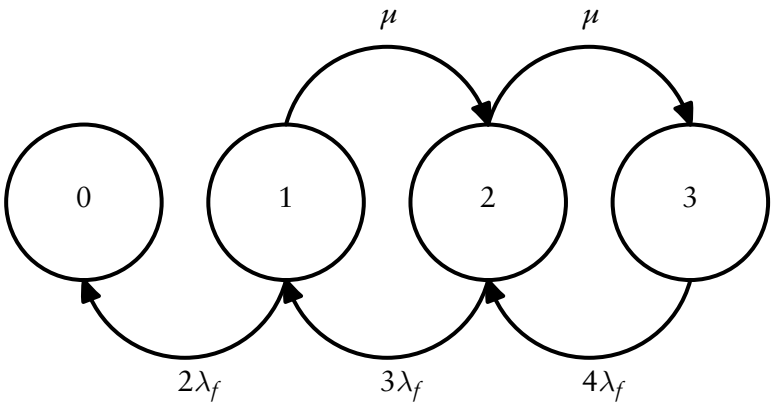


Figure 6-3: A continuous transition, discrete state Markov process that models the behavior of a block's replication level. This model differs from the discrete time model in Figure 6-2 in several ways. First, it is continuous transition: edge weights represent the probability of a transitions in the next Δt instead of per trial. This gives us the ability to talk about probabilities after an elapsed time rather than number of trials. The model is also of finite length to model the fixed r_L used by the algorithms under consideration and has a zero transition probability from state zero (corresponding to the impossibility of recovering lost data).

This gives $P_0 = e^{-\rho}$, where P_0 is the probability that the block is lost.

From this analysis, we glean the intuition that the overriding factor driving data durability is ρ, the ratio between the rate at which replicas can be created and the rate at which they are lost. The average number of replicas is a consequence of this ratio.

Understanding how ρ affects durability helps us evaluate design decisions. For example, when choosing between building a cluster out of n machines with 1TB of storage or $10n$ machines with 100GB of storage, the birth-death analysis tells us that the latter configuration will be more durable since it a replica can be recreated 10 times faster. If the 1TB machines had 10 times the network capacity (1Gb instead of 100Mb Ethernet, perhaps), the two configurations would be equivalent.

This model is arguably more realistic than previous analyses that calculated the probability of loss after a single widespread failure event. The main drawback of the birth-death model is its assumption that inter-failures and recovery times are exponentially distributed. Failures caused by mechanical disk failures might be modeled accurately by the exponential distribution. However, repair times, the rate of temporary failures, and permanent failures caused by operator action can only be modeled approximately by a distribution that specifies memoryless inter-arrival times. For example, repair times are governed by bandwidth and block size and are not likely to vary significantly.

6.2.2 Relationship of r_L to durability

Unfortunately, ρ is largely out of the control of the system: it depends on factors such as link bandwidth, the amount of data stored on each node, and the disk failure rate. The system can control r_L, however, to adjust how aggressively the system replicates blocks. By extending the above analysis, we can evaluate the relationship between different values of r_L, ρ and block durability over time. While it might be possible to design an adaptive algorithm which adjusts r_L based on an estimate of ρ, we only intend to present an analysis that allows system designers to make an informed decision about the appropriate value for r_L.

This analysis will be based on the continuous-time Markov model pictured in Figure 6-3. This model captures the effect of a system that maintains a maximum number of replicas

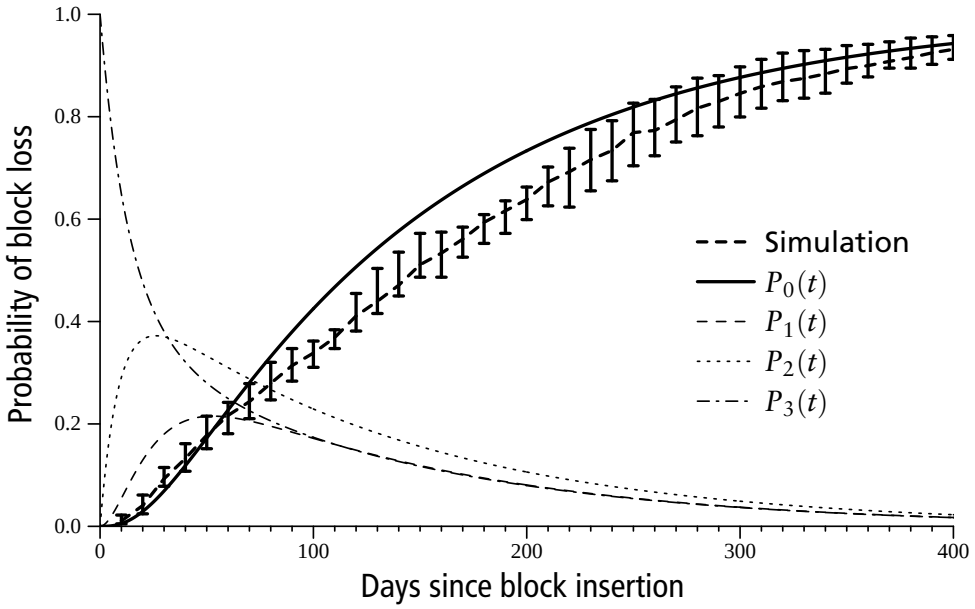


Figure 6-4: Analytic results for the probability of data loss over time. These curves are the solution to the system of differential equations governing a continuous time Markov process that models a replication system that stores 500GB on each PlanetLab node. At time zero, the system is in state 3 (three replicas) with probability 1.0 (dot-dash line is at 1.0). As time progresses, the system is overwhelmingly likely to be in state 0 which corresponds to block loss (that probability is shown as a bold line in the above plot). The rate at which the system moves to depends on the ratio of μ to λ_f as well as their absolute values.

(r_L) by including only r_L + 1 states. Whenever a disk failure causes the number of available replicas to fall below r_L, the system starts creating new replicas of the object until either r_L is reached or the last replica fails. This behavior is modeled by transitions from lower numbered to higher numbered states at rate μ. This (μ) is the rate at which the redundancy lost during a disk failure can be repaired and is determined by factors that include the amount of data stored per node and the capacity of nodes' access links. There is no transition out of state S_0, representing the terminal state where the object has been lost.

Each node fails at a rate λ_f; the rate at which an object experiences failures depends on the number of existing replicas since replicas fail independently. When r replicas exist, each object sees $r\lambda_f$ failures per unit time. However, the rate of transition from the r to the $r-1$ state is modeled as $(r+1)\lambda_f$. The additional factor of λ_f accounts for the fact that the node receiving the new replica could fail while the copy is being transferred (in addition to the r replicas). This "extra" factor causes the model to conservatively assume that a node gets only one chance to make a copy before failing; this assumption makes the model more accurate for environments where the repair time is a large fraction of the failure time and less accurate otherwise. Figure 6-3 shows an example chain for $r_L = 3$.

This chain is a continuous-transition (or continuous-time) Markov process: instead of considering the probability of a transition to a new state in discrete steps, we will define the transition probabilities during the next Δt seconds and then consider the system's behavior in the limit of Δt going to 0.

The probability that the system is in a given state at any time can be expressed as a system of non-homogenous linear differential equations [39]. The solution to these equations gives $P_r(t)$, the probability that the system is in state r at time t. This system can be solved using eigenvalue decomposition, but does not have an easily expressible closed form, so we solve

the system numerically using Mathematica's NDSolve [126].

We can use this model to estimate the probability of block loss given the parameters of the system (μ, λ_f etc.). Because the model assumes memoryless failures we will be predicting the probability of loss due to "coincidental" failures among independently failing replicas. The model does not predict the likelihood of correlated failures such as those caused by operator action.

To apply this model to a particular system, we need to determine values for μ and λ_f. The latter we estimate based on the historical failure record for disks on PlanetLab. Table 2.1 shows the failure characteristics of PlanetLab as derived from the CoMon database; the median failure inter-arrival time for the entire test bed is 4.63 hours, multiplying by the number of nodes in the system (409) and inverting gives an average disk failure rate of $\lambda_f = 1/1900 \approx 0.00053$, failures per hour. The replica creation rate (μ) depends on node storage capacity and network throughput. We'll assume 500GB disks and network capacity that is rate-limited to 1.5Mb/s (the current PlanetLab imposes a 5GB disk quota and a network limit of 1.5Mb/s). Since it takes 740 hours to send 500GB at 1.5Mb/s, the creation rate is $1/740 \approx 0.00135$.

Failure and creation rates are calculated for entire disks, not for single blocks. This is appropriate since data is lost at the granularity of disks. Were we to repeat the above calculations with the rate of block loss and block creation time we would model a different system: one where blocks can be lost individually. Such a system would have a higher probability of block loss for a given replication level (λ_f is larger).

The value of (ρ) for this system is small (around 2.5): we expect that this system will be unable to store data durably. When evaluating the size of ρ, a good rule of thumb is that ρ should be greater than 10. The example system cannot support even a modest number of replicas (no more than 2.5) because it is unable to create replicas much faster than they are destroyed. We'll also consider a hypothetical deployment of UsenetDHT on PlanetLab with a smaller per node storage: 50 and 5 GB. The ρ for these configurations is proportionally larger (25 and 250) and we expect that they will be more durable.

The bold line in Figure 6-4 shows the probability that a block is lost over time for the 500GB configuration. We can interpret this graph as the probability that single block is lost, but note that an entire replica worth of blocks share the same fate. This probability rises towards one as time increases even though the system can create blocks faster than they are lost ($\rho = 2.5 > 1$). Block loss is inevitable in any system given enough time: the system will inevitably experience a series of r_L failures between repair actions. We'll use this analysis to choose an r_L that gives an acceptably small chance of block loss in a bounded time after the block is inserted. The dashed bold line shows the results of a simulation of the same system using a generated trace with a mean disk lifetime matching that observed on PlanetLab. The observed fraction of objects lost at each point in time is plotted (error bars show the minimum and maximum from five runs of the simulator).

The rate at which the probability of block loss increases depends on both ρ and the values of μ and λ_f. As node lifetimes become shorter (the number of failures per unit time increases), the probability that a block is lost increases more rapidly. This result contrasts with our earlier model, which predicted that reliability after a given number of trials depended only on ρ. The value of λ_f can be thought of as the clock for that discrete process. Larger values of λ_f cause more trials per unit time and more chances for blocks to be lost. Note that changing the unit in which the block loss or creation rate is expressed does not affect the outcome. For instance, moving from losses per minute to losses per hour gives the same probability as long as we evaluate $P_0(t)$ at the appropriate t value (e.g. $P_0(360m) = P_0(6h)$).

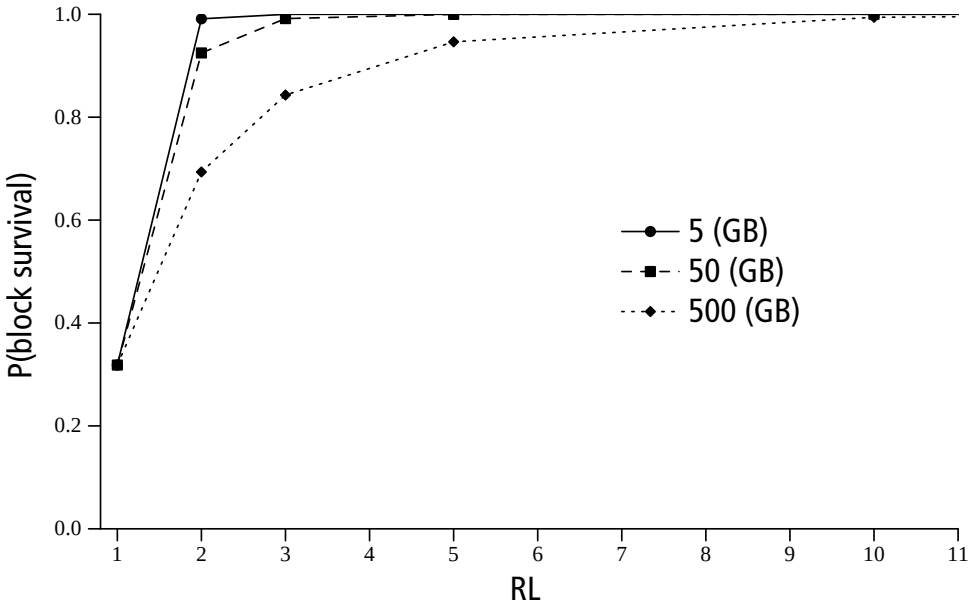


Figure 6-5: Analytic prediction for probability of block durability after 90 days on PlanetLab. The x-axis shows the initial number of replicas for each block: as the number of replicas is increased, block durability also increases. Each curve plots a different per-node data capacity; as capacity increases, it takes longer to copy data after a failure and it is more likely that data will be lost due to multiple simultaneous failures.

The probability of data loss depends on r_L, μ, λ_f, and t. By fixing t, μ, and λ_f, we can consider reliability in terms of only r_L and ρ. We are interested in the relationship between $r_L$and block reliability at a given ρ. Knowing this relationship should help the system designer make an informed decision about how to set r_L. As an example of how to to make that decision, we'll choose parameters based on the PlanetLab environment that currently hosts the UsenetDHT prototype. We'll consider the probability of block loss after 90 days: this is a reasonable amount of time to maintain Usenet articles (and longer than current commercial Usenet hosting services retain binary postings). Figure 6-5 shows the probability that data survives for 90 days for different per-node capacities plotted against r_L. This data is obtained by evaluating $P_0(t)$ in the continuous time Markov model with r_L states; each value of r_L requires evaluating a different model. Each curve rises towards the right: as $r_L$increases the system can tolerate more simultaneous failures and data is more likely to survive. The predicted durability drops as per-node capacity is increased: when more data must be copied after a failure, the window of vulnerability for a simultaneous failure to occur also increases. The probability of block loss at r_L=1 corresponds to using no replication. This value is the same for all curves since it depends only on the lifetime of a disk; no new replicas can be created once the only copy of the data is lost.

From this analysis we can conclude that our current UsenetDHT application, that stores less than 5GB per node, can run safely with 2 replicas (probability of block loss after 90 days is less than 1 percent). If PlanetLab allowed us to store 50GB of data without raising the network ceiling, we might consider using 3 replicas to maintain a 99 percent chance of storing data durably. Storing 500GB per node seems unreasonable given the limited access link bandwidth available on the current testbed.

This analysis underestimates data durability by not considering the effect of temporary failures. If some nodes fail, but return with data intact, the effective number of replicas will

be greater (as long as the system is able to reintegrate returning replicas into the replica sets). However, the analysis overestimates durability by not modeling correlated failures.

6.3 Improving durability

In the previous section we saw that the probability of block loss due to permanent failures depends mainly on ρ, the value of which is largely out of the control of the system, and r_L. In this section we'll consider how to achieve the best durability from a given r_L. The analysis above considers how many replicas of each block are created, it does not specify how those replicas should be distributed across the nodes in the system. This choice has ramifications for data durability and implementation feasibility. By placing replicas to minimize reconstruction time after a failure, we can increase ρ and improve durability.

We consider two possible data placement strategies here that represent the extremes of the design space: random and successor list. Under random placement, each replica is assigned to a random node in the system; a master node maintains a replicated directory that contains the locations of all replicas. The alternative is successor list placement; in this scheme, the r replicas are placed on the r immediate successors of a key in a consistent hashing ring.

6.3.1 Static reliability

Successor placement reduces the probability that any block is lost when some fraction of the nodes in the system fail when compared to random placement. Assume that they ring holds a very large number of blocks (in this case, very large means greater than $\binom{N}{r}$; we'll see why in a moment). Consider what happens when r nodes in the ring fail. Under successor placement, the r failed replicas are unlikely to constitute the entire replica set of any block since that only occurs if they are r adjacent blocks on the ring (an unlikely occurrence). Under random placement, by contrast, some block is very likely to have selected those r nodes as its replica set; that block will be lost. Successor list placement is less likely to lose data in this case because it creates fewer replica sets compared to random placement: N vs. $\binom{N}{r}$. Fewer replica sets implies fewer ways to lose data and a lower probability of data loss given a fixed number of disk failures.

In the rare event that a failure does occur in successor list placement, a large amount of data is lost: $\frac{1}{N}$ of the blocks in the system. The same failure under random placement is likely to lose only $O(1)$ blocks given our assumption on the number of blocks. In fact, the expected number of blocks lost during a failure is equal for both placement strategies; the variance in the number of blocks lost is much higher for successor list placement however. Figure 6-6 plots the distribution of the fraction of blocks that survive the failure of 500 nodes in a 1000 node system. Both distributions have the same median since the expected number of lost blocks is identical (and the distributions are symmetrical), but the variance of the distribution that arises from using successor list placement is larger.

Under this criteria (resilience against a single, massive failure), successor placement seems to be the superior strategy. Others have come to this conclusion [113]. However, random has a significant advantage when a continuous set of failures is considered.

6.3.2 Reconstruction parallelism

The above analysis only considered the system's response to r simultaneous failures; we are more concerned with the system's ability to respond to a continuous process of failure and

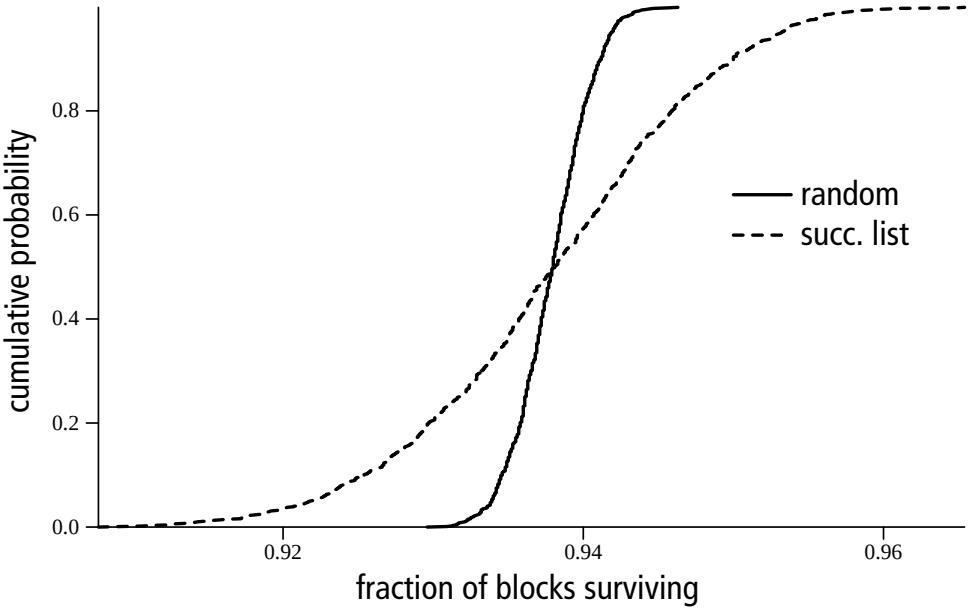


Figure 6-6: The effect of replica placement on block availability following a massive failure. The above plot shows the number of blocks that survive the failure of 500 nodes in a 1000 node system when each block is replicated 4 times. 1000 random trials of the experiment are performed and the CDF of block survival for both successor list placement and random placement is plotted. While both distributions have the same median (the expected number of blocks lost for both placement schemes are identical, successor list has a larger variance.

recovery. Placement strategy affects recovery time by controlling how much parallelism is available to recover blocks.

Consider the replica set of a single block under successor placement. When a node fails, new replicas of each block stored on that node's disk must be created. All of those replicas will be stored by one node: namely the $r+1$st successor of the failed node. Thus, one access must transfer all the copies ($r-1$ nodes behind $r-1$ access links can send copies, but the single access link of the receiving node is the limiting factor).

Under random placement, however, the set of nodes storing replicas of the blocks lost during a disk failure will likely include every node in the system. Additionally, the destinations of the replicas created to replace the lost replicas are chosen at random from all nodes in the system: the set of nodes chosen is again likely to include all nodes in the system. Thus, copying the data blocks will involve N nodes and N access links and thus complete N times faster (assuming no shared bottlenecks among the links). Random placement is able to make a new copy of a failed node's blocks in $1/N$th of the time required by successor placement. This parallelism increases ρ and allows the system to tolerate a higher average failure rate.

Successor placement and random placement are two extremes in a spectrum of placement design choices. More generally, a system could choose to spread r_L replicas among R hosts. This configuration would allow $R - r_L$ access links to download new replicas in parallel. One way to achieve parallelism between 1 and N is to relax the successor list placement to allow replicas to exist an any r_L nodes in the successor list (instead of only the first r_L). When a repair is necessary the successor schedules repair actions from a randomly selected successor holding a replica to a randomly selected successor that does not hold a replica. DHash supports relaxed placement within the successor list and maintains the information necessary to perform parallel repairs but does not schedule the repairs from random nodes.

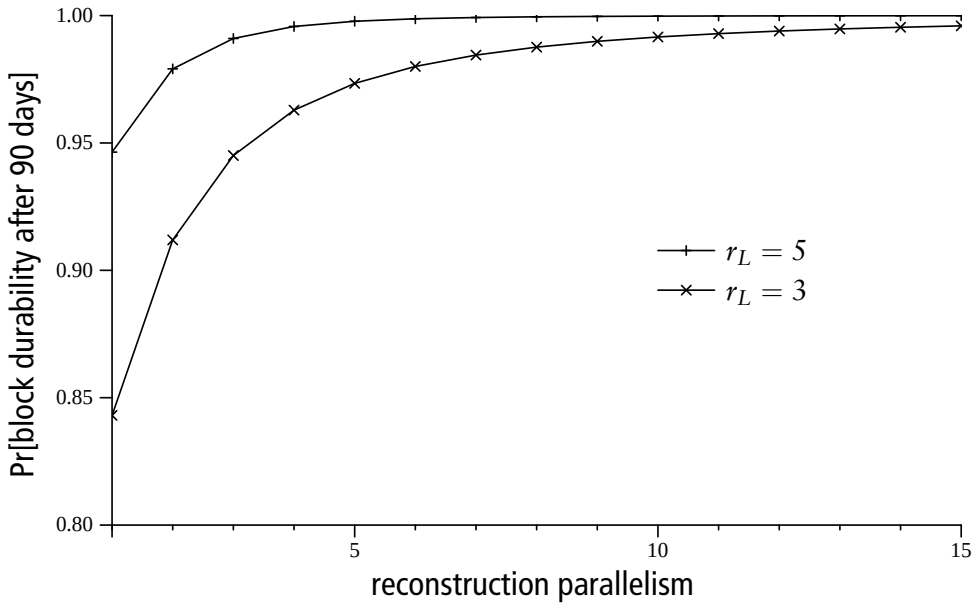


Figure 6-7: The effect of scope on the probability of block loss

To demonstrate the effect of additional parallelism on block loss we reconsider our analysis of block loss probabilities in Section 6.3. Figure 6-7 shows the probability of block survival after 90 days plotted against the amount of reconstruction parallelism for two replications levels (r_L= 3,5). This analysis uses the same failure rate, link capacity, and data scale (500GB per node) assumptions as the analysis presented in Figure 6-5. The points at $R = 1$ corresponds to successor list placement; these values are identical to the values corresponding to $r_L = 3, 5$ in Figure 6-5. As additional parallelism is obtained, the probability of block survival increases. On this plot, the survival probability given by random placement would be off the right side of the graph at $R = N$, where N is the number of nodes in the system. In practice, a system can not achieve a parallelism of N since that would require that failures never occur simultaneously.

The curves in this plot rise approximately as $1 - e^R$. This is not surprising since increasing reconstruction parallelism increases ρ and the probability of block loss is roughly $e^{-\rho}$.

Robbert van Renesse et al. observe that the additional reconstruction parallelism improves block reliability [129]. Their analysis, however, assumes that the number of blocks is small (kN which is less than $\binom{N}{r}$); this grants random placement an additional advantage: at large N, a failure of r nodes is unlikely to cause data loss. This assumption makes it appear that systems with a large number of nodes are less likely to lose data than systems with a small number of nodes; were the number of blocks to be scaled proportionally to the number of replica sets, larger systems would be more likely to lose some data since they create more ways for nodes to fail. Since storage capacity scales linearly with the number of nodes (assuming, for example, that each machine contains a constant number of disks), the small-key assumption may be reasonable.

6.3.3 Data placement: discussion

While the additional reconstruction parallelism that random placement provides is a major advantage, probing overhead may make it hard to implement random placement in some

deployments. Under the assumption that the number of blocks is large, a system employing random placement will be required to periodically contact every node in the system to verify that replicas are stored correctly. Many of the systems we expect to use these replication protocols are designed around the assumption that it is infeasible to maintain information about the entire membership of the system.

In addition, successor placement imposes a structure on the IDs of blocks that are stored on any node (namely that they fall in the range between that node and its predecessor). Section 6.6.2 will discuss a key synchronization algorithm that takes advantage of this structure to efficiently determine which blocks must be transferred between a pair of nodes.

6.4 Reducing the cost of temporary failure

Maintaining r_L replicas is sufficient to provide good durability; this section discusses an additional mechanism that reduces the number of copies of data that must be made in response to temporary failure. Since the goal of DHash is to ensure durability, we consider any replica created in response to a temporary failure to be a mistake. The goal of this section will be to minimize the number of new copies made in response to temporary failures.

We borrow Total Recall's [10] insight that additional replicas beyond the number required for durability (r_L) can reduce maintenance cost. To see why additional replicas help, consider a simple example where r_L is 2. After some time one of the replicas suffers a temporary failure and a third replica is created. At this point, if the failed replica returns and is reintegrated into the replica set, before another failure, the system will not be required to create an additional copy when the next failure occurs.

This savings depends on the system's ability to "remember" the location of replicas that are offline and use them when they return. It is a savings relative to a system that fails to take advantage of the returning node: such a system would be required to make a copy every time one of the replicas becomes unreachable. The magnitude of the savings depends on the number of extra replicas (r_H) that exist and are tracked by the system; if r_H is high enough that r_L replicas are always available, it is unlikely that the system will be forced to create new replicas by future transient failures. An arbitrarily large r_H can be costly, however. If the system decides to create the r_H replicas at block insertion time (as Total Recall does) and the actual number of replicas needed to ensure that r_L replicas are always available is less than r_H, some of the work done to create the r_H replicas was wasted. We'd like to answer the question: how large should r_H be?

We desire an r_H large enough that the system does not need to make new copies of blocks in response to transient failures. In a system that maintains r_L copies at all times, that corresponds to an r_H large enough such that the probability that less than r_L copies are online is low. The following analysis will show that around $2r_L/p$ copies are necessary, where p is the probability that a node is available, independent of any other node (e.g., nodes that are alive as often as they are down have $p = 0.5$).

As r_H increases, the probability that $r_H - r_L$ nodes fail simultaneously falls, making it less and less likely that additional replicas need to be created. The number of available copies of a block (B) can be thought of as the number of successes in a Bernoulli process. The distribution of B is then given by the binomial distribution and the probability of needing a new block, given r_H existing copies, is the probability of fewer than r_L successes., which can be found by summing over the binomial. Each term in the following sum is the probability of a blocks available out of r_H where a ranges over the values that cause the system to create

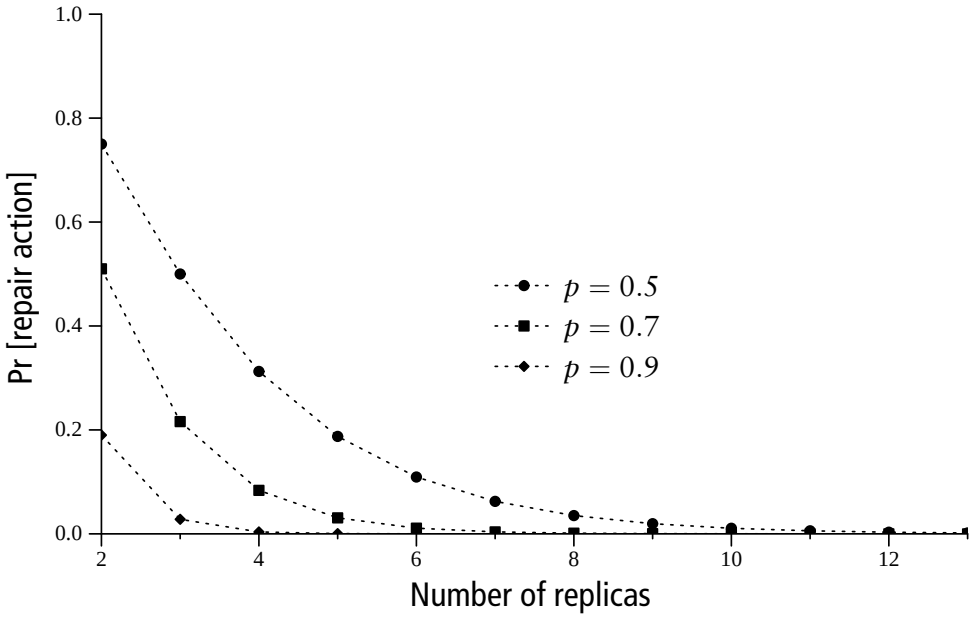


Figure 6-8: The probability of creating a new block falls quickly as more blocks are created. As a result, a system is not likely to create many more than $2r_L/p$ blocks over its lifetime. For $p = 0.9$ and $r_L = 2$ this corresponds to about 4.44.

a new copy $(0 \ldots r_L - 1)$:

$$\Pr[B < r_L \,|\, r_H \text{ extant copies}] = \sum_{a=0}^{r_L-1} \binom{r_H}{a} p^a (1-p)^{r_H - a}.$$

This probability falls rapidly as r_H increases but it will never reach zero; there is always a chance that a block must be created because of a large number of failures regardless of how many replicas are created. When a large number of replicas exist, however, this probability becomes extremely small: $r_H = 2r_L/p$ is a rough (and somewhat arbitrary) estimate of when the probability of a new block creation is small enough to ignore. This value comes from the Chernoff bound on the probability of block creation. From the bound:

$$\begin{aligned} Pr[B < r_L] &< exp(-0.5 r_H p (1 - \frac{r_L}{p r_H})^2) \\ &< exp(-0.5 \frac{(p r_H - r_L)^2}{p r_H}) \end{aligned}$$

When $2r_L/p$ blocks have been created the probability of a block creation is exponentially small: the probability that fewer than r_L of the $2r_L/p$ total replicas are available is less than e^{-r_L}. To see this another way, consider that if $r_H = r_L/p$ the expected number of available blocks is r_L. This r_H is too small: the number of successes will be less than the mean half of the time. A factor of two increase in r_H moves the mean to $2r_L$; it is unlikely that the number of successes will be so far from the mean.

Figure 6-8 shows the probability of block creation after a given number of replicas have been created in a system with $r_L = 2$. Curves are plotted for three different node availability

```
// Iterate through the database and schedule
// keys for repair if needed
MAINTAIN_REPLICAS ()
  foreach k in DB.keys
    n = replicas[k].num_available ()
    if (n < r_L)
      create_new_copy (k)
    else
      // do nothing


// communicate with neighbors and track
// number of available replicas for all keys
SYNC_KEYS ()
 (See Section 6.6.2 for details)
```

Figure 6-9: The algorithm for maintaining extra redundancy.

levels. The probabilities in the figure are over an "experiment" in which all of the replica states are assumed to be randomized. In reality, replicas change state one at a time as nodes crash and return to the system. This discrepancy is a limitation of the analysis, but not a crucial one: it is still able to tell us how large r_H must be to reduce the number of copies caused by node crashes.

This analysis shows us the "penalty" that DHash pays for its inability to distinguish temporary failures from permanent ones. A system that did know which failures were permanent would only need to create r_L replicas. We've shown that it's possible for a system that can't make this distinction to create about r_L more copies of each block over the block's lifetime. This constant penalty is much less than that paid by a naive system that creates a new block in response to every temporary failure: the cost of such a system grows continuously over time.

6.4.1 Coping with transient failure

In Section 6.4 we showed that if the system creates around $2r_L/p$ total copies of a block, it is unlikely that it will need to create additional copies due to transient failures: at least r_L copies will be available at all times with high probability.

Creating the correct number of extra replicas is important. Creating too few extra replicas causes ongoing unnecessary replica creation. With too few copies extant, it is likely that less than r_L machines holding copies will be available: each time this happens a new replica will be created. A system could find itself in this situation if it, for example, failed to re-integrate the replicas on returning nodes into the appropriate replica sets. Creating too many blocks incurs an unnecessarily large insertion cost.

We've already shown that the "right" number of replicas is $2r_L/p$, but without knowledge of p, the system can not use this result directly. In this section we show a practical algorithm (Sostenuto) or creating the correct number of additional replicas without knowledge of node reliability.

Figure 6-9 shows this algorithm. When Sostenuto first creates a block, it makes r_L replicas. Sostenuto monitors the number of reachable replicas of each block. If the number of reachable replicas falls below r_L, Sostenuto eagerly creates new copies to bring the reachable number up to r_L and begins to monitor those nodes as well. This algorithm is equivalent to $r_H = \infty$.

This algorithm produces the correct number of blocks without knowledge of p. After some time Sostenuto will have created a number, B, of copies beyond r_L; these copies make it extremely unlikely that a transient failure will cause a new copy to be created. A new copy could be created only when B nodes are simultaneously unavailable. We showed previously that after $2r_L/p$ total replicas exist it is unlikely that the number of available replicas will drop below r_L; thus it is also unlikely that Sostenuto will create additional blocks. So, we expect the total number of blocks created to be around $2r_L/p$ (Sostenuto will have created $2r_L/p - r_L$ of those replicas).

The effect of these additional replicas is similar to Total Recall's lazy repair [10]. Total Recall specifies a low and high water mark; initially, the system creates a number of replicas equal to the high water mark. When the number of replicas reaches the low water, Total Recall creates enough replicas to return the number of replicas to the high water mark. A poorly chosen high water mark leads to unnecessary network usage. Specifying r_H is difficult; doing so correctly would require knowing p in advance. Sostenuto, unlike Total Recall, does not require the user to specify a high water mark.

An additional advantage of Sostenuto (compared to Total Recall) is that it only creates replicas as they are needed: Total Recall creates its additional replication immediately. Sostenuto creates the minimum possible number of copies necessary to maintain r_L copies at all times since it only does work when the replication level falls below r_L: any algorithm that works by creating an initial buffer of blocks will do at least as much work as Sostenuto. If blocks are created in advance of when then they are needed to bring the replication level above the low-water mark, they are subject to being lost to permanent failures before they become "useful", that is, before they "count" towards bringing the replication level above r_L.

Finally, Sostenuto never forgets about copies of data that it has created. Total Recall will not take advantage of replicas that are unavailable when a repair is triggered even if those replicas were only failed temporarily.

6.5 Evaluation

In this section we evaluate the performance of Sostenuto in simulation using traces of failures on PlanetLab and synthetic traces. The primary metric for the algorithm's performance is the number of bytes sent over the network. We desire an algorithm that minimizes the number of bytes sent over the network while maintaining durability. All of the systems evaluated here more than 99 percent of the blocks were stored durably at the end of the trace.

We will compare the performance of Sostenuto with Total Recall, which represents the state of the art in replication. Total Recall is parameterized by a low and high replication level. We will refer to a Total Recall system with parameters l and h as TR(l, h). Total Recall creates h replicas of a block when the block is inserted; a master node stores an inode-like structure that tracks the current location of the copies of the block. If the replication level of the block ever falls below l repair is initiated and enough new copies of the block are made to bring the replication level back to h. During a repair, the locations of any copies that are

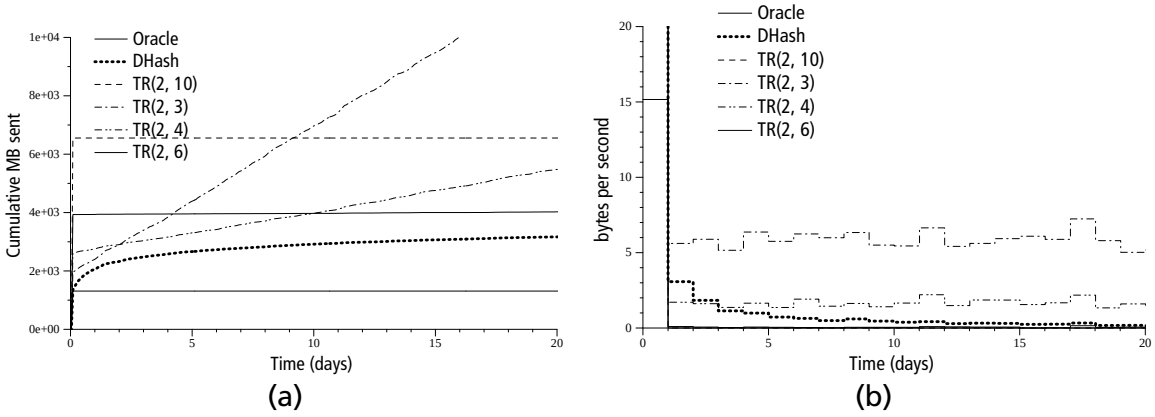


Figure 6-10: A simulation of block repair cost using a synthetic failure trace with no permanent failures. Figure (a) shows the cumulative number of bytes sent to maintain data by each configuration simulated; (b) shows the daily average bandwidth. Since there are no permanent failures, Sostenuto's cost of maintaining data falls to zero over time. In this simulation, node MTTRs are exponentially distributed around 3600s and MTBF is exponentially distributed around 18000s

unavailable at the time of the repair are removed from the inode. Even if the unavailable copies return to the system they will not count towards the replication level of the block.

We first consider the performance of Sostenuto in simulation using the failure traces described in Section 2. We will also consider the performance of the system on a synthetic trace where node and disk failures are the result of independent random processes. In particular we'll generate exponentially distributed failure inter-arrival times.

6.5.1 No disk failures

In this section we consider a synthetic trace with no disk failures. This trace will help us understand the importance of taking advantage of returning copies of blocks to reduce the bandwidth spent creating copies in response to transient failure. We'll compare the performance of Sostenuto against Total Recall with a variety of low and high water marks.

Figure 6-10 (a) shows simulated results for the number of bytes sent by DHash using Sostenuto and Total Recall with a variety of high-water mark parameters. The x-axis of the plot shows time in days and the y-axis shows the number of bytes sent by all nodes in the system up to that time.

Each node in this simulation has an exponentially distributed lifetime with a mean of 24 hours. The down time of the node is also exponentially distributed with a mean chosen to give a node availability (p) of 90 percent.

In the beginning of this simulation, 2500 data blocks are stored; each data block is 768KB (large data blocks allow us to simulate a system that stores a large amount of data quickly: the simulator's runtime is largely determined by the number of blocks). The cost of inserting 2500 blocks depends on the parameters of the system: this is reflected in how high each curve rises at time 0. For instance, Total Recall with a high-water mark of 10 requires 5 times more initial insertion cost when compared to DHash with two replicas.

From the understanding of the costs of replication due to transient failure (see Figure 6-1), we expect that the cost of dealing with transient failures should fall to zero over time if the replication system takes appropriate steps. This can be seen most clearly in Figure 6-10(b) which plots the byte sent per simulated day for each of the systems. In this experiment, DHash and Total Recall with a high water mark of 6 both use nearly zero bytes per second

by the end of simulation.

Total Recall with a high water mark of 3, however, does not see a reduction in the number of blocks created as time goes on. In Section 6.4 we showed that once $2r_L/p$ blocks have been created, it is unlikely that an additional block creation will be necessary. In this scenario, $2r/p = 4/.9 = 3.6$. This explains why TR(2,3) continues to see large numbers of block creations: it is likely that two of the three blocks that TR(2,3) maintains will periodically fail, causing a repair event. Since the blocks created by these repair events are "forgotten" by the system, this process of continual repair will go on forever. This corresponds to the continuous rise of the line corresponding to cumulative bytes for TR(2,3) in Figure 6-10 (a); also, the bandwidth use of TR(2,3) never reaches zero in Figure 6-10(b). Statically configuring Total Recall with too low a high water mark results in unnecessary, continuous bandwidth usage.

The remaining TR configurations and DHash, however, create and remember more than 3.6 blocks. Since no blocks are lost to permanent failure, we can compute DHash's replication level from the number of bytes sent. At the end of the simulation DHash had created 4 replicas per block which is near the 3.6 that our analysis predicts. The number of replicas created is slightly more than 3.6, since $2r_L/p$ only represents the point at which block creations become "unlikely": since blocks are never lost, the replication level will slowly rise as long as the system is running.

TR(2,6) creates and tracks more than 3.6 blocks and also sees low costs due to transient failures. In the case of TR(2,6) the initial number of replicas was so large that the system was never forced to repair a block. While this means that ongoing bandwidth usage is low, it also means that the replicas created initially beyond the approximately 3.6 that were actually necessary to maintain availability were never used. Configuring Total Recall with a too large high water mark results in a one-time waste of bandwidth when blocks are inserted.

The flat, bottom-most line in this plot is the result of a system that uses an oracle to determine which failures are permanent and only creates new copies in response to those failures. Because all of the failures in this experiment are temporary, this system behaves ideally: it produces no copies after the initial insertion. The penalty that DHash pays for not distinguishing permanent and temporary failures is the difference between the cost of the oracle and the cost of DHash.

6.5.2 With disk failures

We now consider the same experiment, but modify the simulator so that 5 percent of node crashes result in data loss. The results are shown in Figure 6-11. In this experiment, permanent failures induce continuous replication work for both DHash and Total Recall. This work is visible as the constant slope of the cumulative bytes curve and the non-zero baseline for the bandwidth curves.

In the experiment with no failures TR(2,6) was able to use initial replicas to avoid doing any repair work for the remainder of the trace. In this experiment, however, the initial replicas are quickly eroded by permanent failures and TR, like DHash, is forced to start creating new replicas. The cost of running both DHash and Total Recall is dominated by the rate of disk failures. TR(2,6) performs additional work at each failure, however, because it maintains a larger number of replicas on average than DHash and thus sees a proportionally higher failure rate. Because TR(2,6) puts more replicas "in harm's way" it loses more data and thus uses more network resources in the long run.

When Total Recall is configured to create less than $2r/p$ replicas, as in TR(2,3), it again

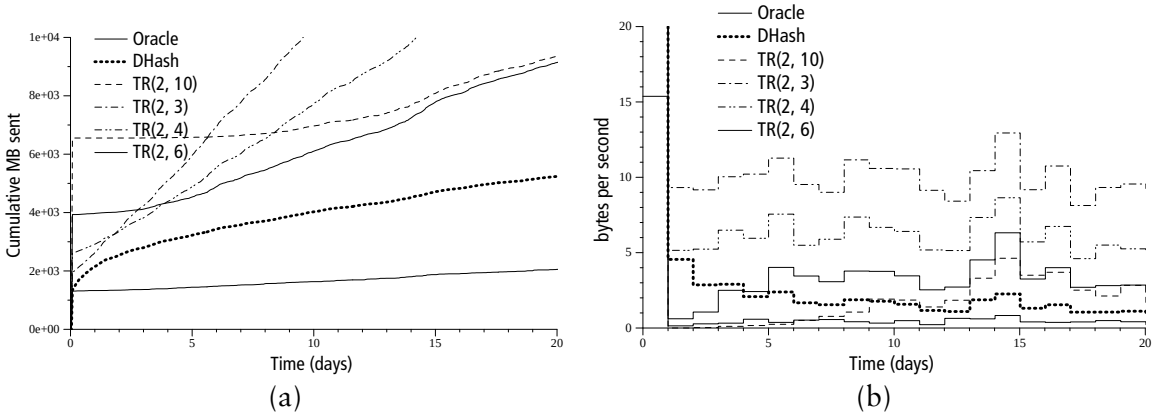


Figure 6-11: A simulation of block repair cost using a synthetic failure trace with permanent failures (5 percent of failures result in data loss). Figure (a) shows the cumulative number of bytes sent to maintain data by each configuration simulated; (b) shows the daily average bandwidth. The permanent failures result in an ongoing maintenance cost.

performs poorly: the cost of repairing transient failures dominates the cost of dealing with the relatively more rare permanent failures.

Disk failures cause the oracular system to do work in this experiment, but it makes fewer copies than the other systems. DHash does more work than the oracular system initially because it reacts to temporary failures. By the end of the trace, DHash and the oracle both create new copies only in response to disk failure. DHash maintains more copies and thus is forced to create new copies at a slightly higher rate than the oracle.

6.5.3 PlanetLab traces

We also evaluated the two algorithms running on a trace derived from failures and crashes of PlanetLab nodes. Figure 6-12 shows the number of bytes sent.

The initial weeks of the trace behave in much the same way as the generated trace without failures. DHash creates about two additional replicas by day 100. Total Recall with a high water mark of 4 or more does not create a large excess of data because, in this part of the trace, the expected number of replicas necessary to weather transient failure is less than 4.

Starting around day 120, disk failures, denoted by vertical lines below the curve showing number of live nodes, begin to drive the bandwidth usage (the are no failures before this point due to limitations of our trace). All schemes are forced to make additional copies in response to disk failures. DHash creates fewer copies than Total Recall during this portion of the trace because it restores the number of replicas only to the low-water mark while Total Recall recreates enough replicas to again reach the high-water mark. Each repair is more costly for Total Recall than for DHash. This effect is most evident at around day 150: a large failure spike causes a large number of repair events. TR(2,10) does more work than the other protocols because it must create a larger number of replicas. In this case, most of that work was unnecessary, the failed nodes quickly returned to the system (note that the oracular system did no work on this day). Because TR(2,10) created the blocks, however, they are now subject to loss. The work was not simply done in advance; because many of the blocks will be lost in disk failures before they can be used by the system as one of the necessary r_L replicas, the work required to create them was wasted. DHash, by contrast, creates new blocks exactly when they are needed.

The net result of this simulation is that DHash causes about 3GB less data to be sent over

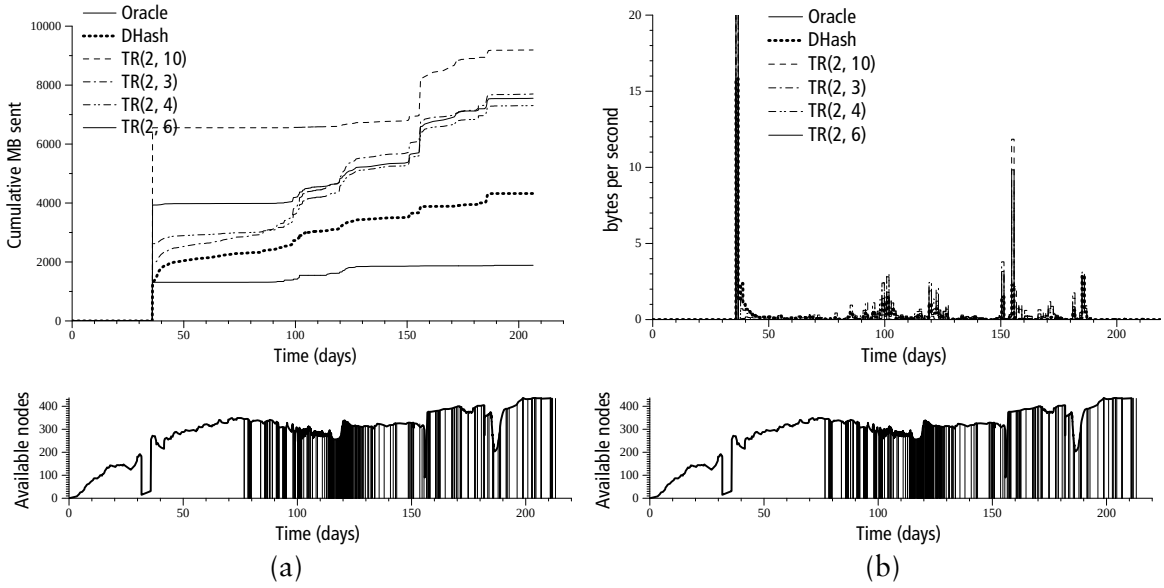


Figure 6-12: A simulation of DHash and Total Recall driven by a trace of PlanetLab nodes. Graph (a) shows cumulative bytes sent; (b) shows the daily bandwidth use of the system. The identical graph below both each plot shows the number of available nodes in PlanetLab (solid line) and the times at which disk failures occurred (thin vertical lines). Disk failures do not occur prior to around day 70 because disk failure information was not available for that time period.

the network when compared to the best performing Total Recall configuration. DHash caused about 3.1GB of maintenance traffic for 1.2GB of replicated data (625 MB of original data): this translates to an additional 5.91 bytes of redundant data sent per byte of original data over the 25 week trace (including the cost of creating the initial redundancy). The overhead of the best performing Total Recall ($r_H = 4$) configuration is 10.7 bytes of redundant data per byte of original data.

The total unavailability of all of these systems is similar. Prior to a massive failure around day 154 of the trace DHash allowed less than 10M block-seconds of unavailability out of the 37 billion block-seconds in the trace corresponding to an availability of 99.97 percent. The over-provisioned configurations of Total Recall provide somewhat better availability: at this point in the trace TR(2,10) suffered no unavailability.

The behavior of all of these systems in response to a large scale, temporary failure (such as the one around 150 days) is not ideal. All of these systems reacted to this event by creating a large number of new replicas; this behavior is evident in the large spike in bandwidth usage at day 150. The oracular system, knowing that the outage was temporary, did not attempt to repair these failures. Creating a non-oracular system that does not react poorly to large temporary failures is the subject of future work (see section 8.3.3).

6.6 Implementation

Sostenuto has been implemented as part of DHash. This section discusses implementation details.

6.6.1 Coding

We have discussed replication in the context of replicating entire blocks. In practice, DHash uses erasure coded fragments. Erasure coding reduces maintenance and storage costs and improves insert throughput when compared to replication. Our analysis of replication still applies when coding is used with minor modifications. $r_L$should now be interpreted as the number of fragments the system desires to keep available (in DHash this value is currently 14).

When calculating block loss probability the Markov model must be modified to take into account the fact that block loss occurs when fewer fragments than are needed to reconstruct a block remain. In particular, this increases the rate at which fragments are lost since more fragments exist. This effect alone would make it more likely that blocks will be lost; however, since a system can "afford" more fragments than full replicas at the same overhead, a fragmented block is less likely to be lost than a replicated block when the same number of bytes are dedicated to replication.

Using coding also causes the cost due to transient failures to fall more quickly when compared to whole-block replication. Not only are nodes exchanging smaller objects during repair, but the fact that more failures must occur to trigger repair makes repairs less likely (1 failure out of 2 is far more likely than 7 out of 14).

When using coding, systems must be careful to amortize the cost of fragment creation since creating a fragment requires fetching and reconstructing a block. A naive system will spend more bandwidth fetching blocks than it saves by sending out smaller (than full replica) fragments. Caching the reconstructed block following a reconstruction and creating subsequent fragments from the cached block largely eliminates this problem.

The benefits of coding have been discussed elsewhere [135, 35]. We won't discuss it in further detail in this chapter.

6.6.2 Synchronizing Keys

While we have focused on the cost of making new copies due to failure, the costs associated with monitoring data between failures cannot be ignored. A simple approach to monitoring would be to assume that a node's availability implies availability of any data it has previously been known to store. This approach is inexpensive (the cost grows only with the number of nodes that must be probed) but incomplete: database corruption or configuration errors could result in continued uptime of a node while some or all stored data blocks are lost.

We'll outline the design of a system in which the node responsible for a given key wishes to make an end-to-end check that data is stored on disk. This master node wishes to determine which replica nodes are not holding copies of the data item and which data items are held on other nodes that the master should be responsible for. The latter case occurs when failure causes a node to become master for a new set of data.

DHash uses a synchronization protocol based on Merkle trees to efficiently take advantage of the fact that most blocks are typically in the right place [19]. This scheme is much more efficient than simply exchanging lists of 20-byte keys; in the common case where a replica node holds copies of all desired keys, the protocol can verify that all keys were stored with a single RPC. This protocol depends on the successor placement scheme.

However, the current implementation of DHash only requires that the r_L replicas be located in the successor list of the node responsible for the key. DHash is typically configured to maintain a successor list roughly twice as large as r_L. This flexibility allows nodes to

```
//build a merkle key from database and
//information about where keys are missing
//tree is "customized" for node N
BUILD_TREE (DB db, block_info bi, ChordID N)
    foreach  key in db.keys
            if (!bi.missing_on (N, key))
              insert (key) //insert the key in the tree
            else //leave the key out of the tree

    // iterate over keys the local node is missing and add keys missing on the local node
    foreach lkey in (bi.missing_keys (my_ID ()))
            if (bi.confirmed_present (N, lkey) or !bi.missing_on (N, lkey))
               insert (lkey)
```

Figure 6-13: Pseudocode for the building a Merkle tree based on information about where keys are missing. The first iteration over keys in the database is modified to leave out keys that are known to be missing on the remote node. The second iteration, over keys that the local node is known to be missing, adds keys that the local node doesn't store. The local node "fakes" storing keys that it is missing but that it knows to be stored on the remote node or those keys for which it is unsure of the remote node's status (by adding the key tree, the remote node will report whether or not it stores the key during synchronization).

join the system without transferring large amounts of data as they would if the placement of replicas was rigidly enforced. This policy has the natural effect of placing more data on more reliable nodes. It also allows for the possibility of increased reconstruction parallelism (see Section 6.3.2).

Unfortunately, this flexibility causes the Merkle synchronization protocol to operate outside of its common case: adjacent nodes are no longer likely to store nearly identical sets of blocks. The result is that each time the synchronizer runs, it is forced to transfer a list of keys. This transfer can be costly. If the synchronization protocol runs once a minute, the cost of repeatedly transferring the 20-byte name of an 8KB data block will exceed the cost of transferring the block itself to a newly joined node in about 8 hours. This problem is more severe when erasure coding is used since blocks are smaller: the cost of transferring associated keys more quickly outstrips the size of the data.

To avoid this problem we enhanced the synchronization protocol to efficiently deal with nodes that are missing blocks. We require the master node to track the identity of nodes missing copies of a given block. Using this information about missing replicas, the master can modify the synchronization protocol to not transfer redundant information. For instance when synchronizing with a replica node n that is known to be missing a key k, the master leaves k out of the tree used for synchronization: this prevents n from reporting what the master already knew, that k is missing on n. This scheme requires the master node to keep additional state about where keys are missing within the successor list. However, this amount of state is small relative to the size of storing the block itself, and can be maintained lazily, unlike the hard state that the node stores for each key (namely the block). Figure 6-13 shows pseudocode for this process.

6.7 Conclusion

Understanding the costs associated with data maintenance helps create a more complete estimate of the costs of DHT-based applications. For example, the initial analysis of the operating cost of UsenetDHT did not include the costs of maintaining data. That analysis, found in [118], estimates a network cost of 1 Mb/sec (compared to 100Mb/s for an average Usenet installation), based on 1.4 TB of postings per day and 300 servers. The main source of the reduction is the use of a DHT to organize the storage of the nodes; using the DHT only a few copies need be stored in the system while Usenet stores a copy at every node.

We can use the above simulations to estimate the maintenance cost of UsenetDHT and gain a more complete understanding of its costs. If the system stores articles for the length of our trace, 25 weeks, the DHT must send 5.91 bytes for every byte inserted over the 25 week period. Since a binary Usenet feed generates about 1.4 TB daily, the nodes participating in the DHT must send $1.4 \times 5.91 \approx 8.27$ Terabytes. Spread over the 300 nodes of a hypothetical UsenetDHT deployment, this requires each node to source about 2.5Mb/s. This number is greater than the initial estimate of 1Mb/s, but still far below the requirements of existing Usenet servers.

This overhead represents a significant improvement over base DHash: we estimate that the maintenance traffic of that system would be approximately 6 times larger.

7

Related work

DHash provides a simple `put/get` storage interface that allows applications to easily utilize the storage resources of machines connected to the Internet. DHash runs in a wide-range of deployment scenarios from machine-room clusters to cable modems and provides eventual consistency. DHash was among the first distributed hash tables that provide a similar interface (PAST, CAN, and Tapestry were developed concurrently with DHash). In this section we discuss these and other related works. We begin with distributed storage systems (Section 7.1), and then consider distributed object location systems (Section 7.2). Section 7.3 discusses Vivaldi's relation to work on network location. Finally, we look at how other systems maintain replicas in Section 7.4.

7.1 Distributed storage

A number of distributed storage systems have been developed to run on local area networks [49, 128, 45, 75, 9, 77, 76]. All of these systems are designed to operate in highly connected, trusted environments such as a cluster of machines in a machine room. They provide high performance, availability, and strong atomicity guarantees while storing large amounts of data. DHash operates in the wide-area environment and differs in design as a result. For instance, DHash has no centralized components and does not run consensus algorithms that have a high round-complexity. As a result of these compromises DHash offers much weaker atomicity guarantees: partitions could lead to two versions of a public key block, for example. DHash's aggregate throughput performance scales with the number of nodes in the system: at large scales this aggregate performance will match that of tightly coupled systems.

7.1.1 File sharing systems

DHash was inspired by early peer-to-peer file sharing applications most notably Napster, Gnutella, and Freenet [87, 48, 26]. These applications export a keyword search interface that is not provided directly by DHash. Of the three, DHash is most closely related to Freenet. DHash represents, like Freenet, a compromise between the centralized architecture of Napster and the flooded queries of Gnutella. Unlike Freenet, DHash, because it uses Chord, provides a tight bound on the number of hops a lookup makes and guarantees lookup success (except in the case of a large number of simultaneous failures). DHash, unlike Freenet, makes no attempt to hide the identity of publishers or clients.

Among recent peer-to-peer sharing systems, BitTorrent [11, 27] bears the most resemblance to DHash. BitTorrent distributes large data items from multiple servers by dividing

them into blocks: any client that has downloaded part of the object can serve those blocks it has stored. A centralized tracker distributes information about what nodes are participating in the system; pairs of nodes exchange information about which blocks they are holding. DHash distributes files as blocks in much the same fashion; because it is built on Chord, however, a centralized tracker is not necessary to maintain membership. BitTorrent replicates data more widely than DHash by using recent downloads as replicas; DHash's caching scheme behaves similarly.

7.1.2 Other DHTs

The research systems most closely related to DHash are other DHTs built on log N hop object location protocols.

Pond [107] is a prototype of the OceanStore [69] system. Pond and OceanStore are based on the Tapestry location system (instead of DHash's Chord). OceanStore has a number of additional features in the storage layer beyond what DHash provides including mutable blocks with versioning, strong update semantics, and Byzantine fault tolerance. We believe that the weaker semantics and simpler interface that DHash provides will be suitable for most, if not all, applications. In addition, this choice has made DHash easier to implement (along with the relative simplicity of Chord). Work on the implementation of OceanStore has been suspended by the authors as of this writing in favor of the Bamboo DHT [108], which has a feature set more in line that provided by DHash.

Bamboo uses Pastry for object location, but with modifications to routing table maintenance that enable it to function under high node failure/join rates ("churn"). Our Chord implementation and Bamboo have similar table update strategies: both should perform well under churn.

The OpenDHT [92, 109, 64] project is a publicly accessible deployment of Bamboo on PlanetLab. Like,DHash the primary goal of OpenDHT is to easy application development. By allowing the public to store data on a collection of servers maintained by the project, OpenDHT hopes to eliminate one barrier facing applications that want to use a DHT: running the DHT itself.

PAST [112] is a DHT built on the Pastry [111] object location system. Unlike DHash, PAST is optimized to store large blocks (whole files), while DHash uses 8KByte blocks. Larger blocks make it easier to obtain high throughput since the latency of a lookup can be amortized over a long download. However, larger blocks make load-balance more difficult. When storing large objects the variance in per-node storage is proportionally higher: it is not unlikely that some node will exhaust its storage resources before the entire system reaches its capacity. PAST implements mechanisms to divert data objects to less loaded replicas. DHash does not need to implement such a mechanism: because it stores small blocks, it is unlikely that one node will reach its capacity before the system as a whole does.

7.2 Object location

DHash is based on the Chord object location system and this thesis describes how that system can be optimized to achieve lower latency. Many other object location systems have been proposed and DHash could be built on many of them. In addition, the latency optimizations we describe can and are used by many location systems.

7.2.1 Algorithms

The distributed data location protocol developed by Plaxton *et al.* [99] is perhaps the closest algorithm to the Chord protocol. The Tapestry lookup protocol [138], used in OceanStore [69], is a variant of the Plaxton algorithm. Like Chord, it guarantees that queries make no more than a logarithmic number of hops and that keys are well-balanced. The Plaxton protocol's main advantage over Chord is that it ensures, subject to assumptions about network topology, that queries never travel further in network distance than the node where the key is stored. Chord, on the other hand, is substantially less complicated and handles concurrent node joins and failures well.

Pastry [111] is a prefix-based lookup protocol that has properties similar to Chord. Pastry uses a smallest numeric difference metric to determine neighbors instead of the consistent hashing successor metric used by Chord.

CAN uses a d-dimensional Cartesian coordinate space (for some fixed d) to implement a distributed hash table that maps keys onto values [106]. Each node maintains $O(d)$ state, and the lookup cost is $O(dN^{1/d})$. Thus, in contrast to Chord, the state maintained by a CAN node does not depend on the network size N, but the lookup cost increases faster than $\log N$. If $d = \log N$, CAN lookup times and storage needs match Chord's.

The Globe system [7] has a wide-area location service to map object identifiers to the locations of moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative domains, effectively constructing a static world-wide search tree, much like DNS. Information about an object is stored in a particular leaf domain, and pointer caches provide search shortcuts [130]. The Globe system handles high load on the logical root by partitioning objects among multiple physical root servers using hash-like techniques. Chord performs this hash function well enough that it can achieve scalability without also involving any hierarchy, though Chord does not exploit network locality as well as Globe.

Kaedemlia is a lookup system that uses iterative lookup, but takes advantage of parallelism to avoid timeouts and find low-latency paths. Kaedemlia routes in an XOR-based metric space; in this space, if node A that has a pointer to node B is its routing table, B is also likely to have a pointer to A. This symmetry is intended to reduce maintenance costs.

Several DHTs store more than $\log N$ state in order to reduce lookup hop-count. Kelips [54] maintains $\sqrt{N}$ state and can perform a lookup in $O(1)$ hops. Kelips usually requires two hops: one to one of $\sqrt{N}$ groups and one more within the group (group members maintain complete state about other group members). The One-hop DHT [53] maintains $O(N)$ state and gives $O(1)$ (usually 1 hop) lookups. Accordion [72] limits maintenance traffic to a user-specified budget, but in a stable system usually collects state about every node in the system, resulting in a 1-hop lookups. All of these systems will provide lower latency lookups than Chord at the expense of additional traffic to maintain routing tables.

Other DHTs maintain less state than Chord and still provide $O(\log N)$ lookups. Koorde [62] is structured around deBruijn graphs and maintains $O(1)$ state. Viceroy [78] has the same routing table and lookup requirements as Koorde.

Chord's routing procedure may be thought of as a one-dimensional analogue of the Grid location system (GLS) [71]. GLS relies on real-world geographic location information to route its queries; Chord maps its nodes to an artificial one-dimensional space within which routing is carried out by an algorithm similar to Grid's.

7.2.2 Optimizing DHTs

A great deal of research has been done to optimize DHTs. Gummadi and Gummadi discuss the impact of different DHT geometries on lookup latency [50]. Their conclusion that flexibility is crucial to achieving low latency inspired our modifications to Chord, specifically the use of proximity neighbor selection (PNS) instead of proximity route selection (PRS). Castro et al. also conclude that PNS provides the best latency reductions and propose alternative methods of building network aware routing tables [16, 17].

The PVC framework [73] evaluates the tradeoff between overlay routing latency and the bandwidth required for table maintenance; this work does not consider further optimizations to probing overhead.

7.3 Network location

A number of systems have been proposed for computing synthetic coordinates. In addition, the mass-spring minimization used by Vivaldi has also been used previously. Unlike any one of the related systems, Vivaldi is fully decentralized, targets the Internet, and adapts well to changes in the network. Vivaldi uses height vectors to improve its latency predictions, while previous systems mostly use Euclidean coordinates.

7.3.1 Centralized coordinate systems

Many existing synthetic coordinate systems involve a centralized component (such as a set of landmark nodes) or require global knowledge of node measurements. Vivaldi's main contribution relative to these systems is that it is decentralized and does not require a fixed infrastructure.

Vivaldi was inspired by GNP [90], which demonstrated the possibility of calculating synthetic coordinates to predict Internet latencies. GNP relies on a small number (5-20) of "landmark" nodes; other nodes choose coordinates based on RTT measurements to the landmarks. The choice of landmarks significantly affects the accuracy of GNP's RTT predictions. Requiring that certain nodes be designated as landmarks may be a burden on symmetric systems (such as peer-to-peer systems). In addition, landmark systems, unlike Vivaldi, do not take advantage of all communication between nodes: only measurements to landmarks are helpful in updating coordinates.

NPS [89] is a version of GNP which addresses system-level issues involved in deploying a coordinate system. NPS includes a hierarchical system for reducing load on the landmark nodes, a strategy for mitigating the effects of malicious nodes, a congestion control mechanism, and a work-around for NATs. NPS also demonstrates how a landmark-based coordinate system can adapt to changing network conditions by periodically communicating with the landmark nodes and recomputing coordinates.

Lighthouse [97] is an extension of GNP that is intended to be more scalable. Lighthouse, like GNP, has a special set of landmark nodes. A node that joins Lighthouse does not have to query those global landmarks. Instead, the new node can query any existing set of nodes to find its coordinates relative to that set, and then transform those coordinates into coordinates relative to the global landmarks.

Tang and Crovella propose a coordinate system based on "virtual" landmarks [127]; this scheme is more computationally efficient than GNP and is not as dependent on the positions of the landmarks. The authors also present an analysis of the dimensionality of

their measured latencies which suggests that low-dimensional coordinates can effectively model Internet latencies. This result agrees with the results presented in Section 3.4

7.3.2 Decentralized Internet coordinate systems

PIC [28], like Vivaldi, can use any node that knows its coordinates as a landmark for future nodes: PIC does not require explicitly designated landmarks. This approach requires that a PIC node either knows or doesn't know its coordinates. Vivaldi incorporates a continuous measure of how certain a node is about its coordinates; this approach helps adapt to changing network conditions or network partitions. PIC does not seem to be suited to very dynamic conditions: it runs the Simplex minimization procedure to completion after a node measures the latencies to the landmarks. This makes PIC react very quickly to new measurements. In building Vivaldi we found that reacting too quickly to measurements (using a large time-step) caused the coordinates to oscillate.

PIC defends the security of its coordinate system against malicious participants using a test based on the triangle inequality. Vivaldi defends against high-error nodes, but not malicious nodes. PIC uses an active node discovery protocol to find a set of nearby nodes to use in computing coordinates. In contrast, Vivaldi piggy-backs on application-level traffic, assuming that applications that care about coordinate accuracy to nearby nodes will contact them anyway.

NPS [89] uses a decentralized algorithm. Instead of sending measurements to a central node that runs the Simplex algorithm to determine landmark coordinates (as GNP does), each NPS landmark re-runs the minimization itself each time it measures the latency to a new node. Vivaldi operates in much the same manner, but uses a mass-spring system instead of the Simplex algorithm to perform the minimization.

SVivaldi [37] is a modification of Vivaldi that eliminates the tendency of the coordinates of all of the nodes in a Vivaldi system to drift together. This drift might be a problem if nodes cached coordinates for a long time. SVivaldi eliminates drift by slowly reducing δ over the lifetime of a node: eventually δ becomes zero and all node stop adjusting their positions. Vivaldi maintains a non-zero δ to let the system continually adjust to changes in the network: the authors of SVivaldi suggest that δ could be reset when a significant change in the network is observed but don't describe a mechanism that detects such a change.

7.3.3 Coordinate systems for wireless nets

The distributed systems described in this section target wireless networks where distance, either in network latency or physical separation, reflects geographic proximity fairly closely. Vivaldi is intended for use in the Internet, where the topology of the network is much less obvious *a priori*.

AFL [101], a distributed node localization system for Cricket [103] sensors, uses spring relaxation. The Cricket sensors use ultrasound propagation times to measure inter-sensor distances and cooperate to derive coordinates consistent with those distances. Most of the complexity of AFL is dedicated to solving a problem that doesn't affect Vivaldi: preventing the coordinate system from "folding" over itself. Unlike the sensors used in the Cricket project, we assume that Vivaldi hosts can measure the latency to distant nodes; this eliminates the folding problem.

Other systems (such as ABC [114]) operate by propagating known coordinates from a fixed set of anchor nodes in the core of the network. Each node can then find its location in

terms of the propagated coordinates using techniques like triangulation.

Rao *et. al.* [105] describe an algorithm for computing virtual coordinates to enable geographic forwarding in a wireless ad-hoc network. Their algorithm does not attempt to predict latencies; instead, the purpose is to make sure that directional routing works. Beacon vector routing creates a coordinate system that predicts the number of hops to a set of landmark nodes; the coordinates are used to perform greedy geographic routing [40].

7.3.4 Spring relaxation

Several systems use spring relaxation to find minimal energy configurations. Vivaldi's use of spring relaxation was inspired by an algorithm to recover a three dimensional model of a surface from a set of unorganized points described by Hugues Hoppe [59]. Hoppe introduces spring forces to guide the optimization of a reconstructed surface.

Mogul describes a spring relaxation algorithm to aid in the visualization of traffic patterns on local area networks [85]. Spring relaxation is used to produce a 2-dimensional representation of the graph of traffic patterns; the results of the relaxation are not used to predict latencies or traffic patterns.

Graph drawing algorithms use spring relaxation to find the position of nodes when planarizing graphs. The goal of these algorithms is visual clarity [8].

The Mithos [131, 110] overlay network uses a spring relaxation technique to assign location-aware IDs to nodes.

Shavitt and Tankel's Big Bang system [115] simulates a potential field similar to Vivaldi's mass-spring system. Their simulation models each particle's momentum explicitly and then introduces friction in order to cause the simulation to converge to a stable state. Vivaldi accomplishes the same effect by ignoring the kinetic energy of the springs. The Big Bang system is more complicated than Vivaldi and seems to require global knowledge of the system; it is not clear to us how to decentralize it.

7.3.5 Internet models

Vivaldi improves its latency predictions with a new coordinate space that places nodes some distance "above" a Euclidean space. Previous synthetic coordinate systems concentrated on pure Euclidean spaces or other simple geometric spaces like the surfaces of spheres and tori.

Shavitt and Tankel [116] recently proposed using a hyperbolic coordinate space to model the Internet. Conceptually the height vectors can be thought of as a rough approximation of hyperbolic spaces. The hyperbolic model may address a shortcoming of the height model; the height model implicitly assumes that each node is behind its own access link. If two nodes are behind the same high-latency access link, the height model will incorrectly predict a large latency between the two nodes: the distance down to the plane and back up. Comparing the height vectors and hyperbolic model directly to determine which is a better model for the Internet is future work.

7.3.6 Other location techniques

IDMaps [42] is an infrastructure to help hosts predict Internet RTT to other hosts. The infrastructure consists of a few hundred or thousand tracer nodes. Every tracer measures the Internet RTT to every other tracer. The tracers also measure the RTT to every CIDR address prefix, and jointly determine which tracer is closest to each prefix. Then the RTT

between host h_1 and host h_2 can be estimated as the RTT from the prefix of h_1 to that prefix's tracer, plus the RTT from the prefix of h_2 to that prefix's tracer, plus the RTT between the two tracers. An advantage of IDMaps over Vivaldi is that IDMaps reasons about IP address prefixes, so it can make predictions about hosts that are not aware of the IDMaps system.

Meridian [137] is a system that answers on-line queries for the nearest node, in the system, to an arbitrary node on the Internet. To answer a query, Meridian routes a query through a network in much the same way Chord does, but routing table entries are chosen based on latency rather than on a position in the ID space: nodes have more pointers to nearby nodes than to distant nodes. Meridian answers queries more accurately than Vivaldi, but at the cost of a long delay in obtaining results. A query in Meridian requires on the order of 300ms to return. This makes the system unsuitable for the tasks Vivaldi is used for in DHash. Meridian might be useful for choosing a nearby server in a content-distribution network since they delay of the lookup can be amortized over multiple requests and the client node is not likely to be able to participate in a coordinate system.

The IP2Geo system [94] estimates the physical location of a remote server using information from the content of DNS names, whois queries, pings from fixed locations, and BGP information. IP2Geo differs from Vivaldi mainly in that it attempts to predict physical location rather than network latency.

7.4 Replica maintenance

This thesis has a proposed data maintenance algorithm, Sostenuto. Many previous systems have used replication to ensure data availability and durability. The main difference between this work and previous work is that previous work assumes that failures are rare enough, or inexpensive enough that it is not necessary to minimize the cost of dealing with failures.

Petal is a virtual disk that replicates data at the block level across two machines. A Petal virtual disk is available during a failure, but operates at a degraded performance level [70]. Like Sostenuto, Petal considers the effect of placement on data reliability: the authors note that the chained declustering placement strategy increases the chance the some data will be lost relative to simple mirroring. Under chained declustering the failure of any two servers causes data loss (analogously to random placement with $r = 2$).

DDS also manages replicated data at the block-level using a primary-backup scheme. Because DDS assumes a fast, local-area connection between storage servers and that failures are rare, it has a simple recovery procedure. During recovery the replica group being recovered is locked and all operations on that group begin to fail. The recovering node then downloads the affected partition. No provision is made for transient failures: all failures are assumed to destroy the media. DDS's recovery strategy is very simple, but is not likely to work in an environment where transient failures are commonplace (as we have assumed).

Harp also performs primary-backup replication but does so on the file-system level. Like Sostenuto, Harp is optimized to ease recovery when a failed node returns with its disk intact. To ease recovery in this case, the witness copies its log to new secondary, rather than transferring the entire file system.

The Google File System [45] is a large scale storage system that operates across thousands of commodity machines located in multiple machine rooms. GFS operates as a directory system with a master for each file that manages replicas of files across many machines. Like systems we target, GFS must deal with unreliable nodes and make intelligent placement and repair decisions. However, GFS has fine-grained knowledge about nodes due to its central

management and single master node: for example, placement decisions can be made to take advantage of rack placement. Further, bandwidth is relatively abundant. The heuristics used by GFS are very similar to those we propose: for example, repair is prioritized to favor chunks that are low on replication. GFS allows the user to set the desired replication level (e.g. to allow for increased parallelism in reads) but defaults to $r_L = 3$.

The above systems employ simple replication and recovery strategies, but can afford to because they are designed to run on the local area network; such strategies would be infeasible in a wide-area setting. Systems like Coda and Bayou operate in a wide-area setting where failure and disconnection are common by providing loose consistency and tolerating conflicts during reconciliation. The consistency model provided by DHash, while not giving strict ACID semantics, is more strict than the eventual consistency provided by Bayou. Sostenuto's replication algorithms are correspondingly more aggressive.

Total Recall is the system most similar to our work [10]. We borrow from Total Recall the idea that creating and tracking additional replicas can reduce the cost of transient failures. Total Recall's lazy replication keeps a fixed number of replicas and fails to reincorporate replicas that return after a transient failure if a repair had been performed. Total Recall must also use introspection or guessing to determine an appropriate high water mark that Sostenuto can arrive at naturally.

The model of block loss and creation we use to predict durability is intended to capture the continuing process of replica loss and repair that occurs in a long-running system. Prior work instead considered failures as independent events and calculated the probability of data loss as the probability that all replicas were simultaneously in the failed state [135, 34, 12]. Such a calculation reasonably captures the effect of single, massive failure (e.g. half of the nodes in the system fail at the same instant). Models based on independent failures can not answer questions about the probability of data loss after a certain amount of time has elapsed, however.

Other systems choose replication levels with the goal of continuing operation despite a fixed number of failures or to provide enough remaining hosts to vote or run some other agreement protocol to maintain consistency following a disk failure [18, 75].

Our birth-death model is a generalization of the calculations that predict the MTBF for RAID storage systems [96]. Owing to its scale, a distributed system has more flexibility to choose parameters such as the replication level and number of replica sets when compared to RAID systems.

8

Conclusion

This thesis has described the design and implementation of DHash, a distributed hash table. DHash makes it easier for programmers to write distributed applications by providing access to the storage resources of a large number of nodes via simple, hash-table interface. Building DHash required overcoming several challenges: to be useful to applications DHash must provide high-throughput, low-latency access to data despite running on many unreliable hosts connected by a high-latency, failure-prone wide-area network.

This summarizes the main results presented in this thesis, considers some design issues that span more than one chapter, and presents future work.

8.1 Summary

This thesis introduced a number of algorithms and techniques to overcome the challenges facing a system that attempts to durably store a large amount of data on the wide-area network and provide low-latency, high-throughput access to that data.

Proximity neighbor selection enables DHash to perform low-latency lookup operations by selectively populating routing table entries with nearby nodes. We showed how this optimizations leads to an expected lookup latency of around $3 \times \delta$ where δ is the median latency in the system.

In finding nearby neighbors (and in a number of other operations such as setting retransmission timers, and choosing nearby servers for retrieving data) DHash is assisted by Vivaldi, a distance estimation system. Vivaldi is a decentralized system that produces synthetic network coordinates: low-dimensional Euclidean coordinates that are chosen such that the distance between two such coordinates is a good predictor of the latency between the corresponding nodes. Coordinates allow DHash to predict the latency to a node without communicating with that node. Many of the optimizations to DHash would have been impossible without the ability of coordinates to make instant latency predictions.

DHash achieves high throughput with the help of the STP protocol. STP provides effective congestion avoidance while holding many short conversations with many unique hosts; stream-oriented protocols such as TCP perform poorly in this environment.

Data replication and maintenance are handled by Sostenuto, a data maintenance algorithm that maintains durability while minimizing the number of copies sent over the network.

Using these techniques is is possible to build applications such as UsenetDHT, run them efficiently, and easily add resources to the deployment as the applications grow in popularity. By reducing the latency of `get` operations it is possible to read news interactively using UsenetDHT. Sostenuto reduces the cost of running UsenetDHT: the base DHash replication scheme would have sent about 15Mb/s of traffic on the network; Sostenuto sends 2.5 Mb/s.

8.2 Tradeoffs

Designing DHash necessitated a number of tradeoffs. Some of these involve concerns that span the chapters of this thesis and can not be addressed in the context of a single chapter. We address those here.

The amount of routing state that Chord maintains sets up a tradeoff between routing latency and maintenance traffic overhead. Chord maintains an amount of routing state proportional to $\log N$. Maintaining more state could reduce lookup latency by eliminating lookup hops but would increase the amount of network traffic necessary to maintain the routing information: other DHTs have taken this approach [53, 54]. Still other DHTs maintain even less state than Chord [62, 78]. Chapter 4 did not consider the cost of maintaining routing tables; we safely assumed the cost to be small since even in a large system the number of routing entries that Chord maintains is small. A thorough analysis of the tradeoff between maintaining more routing state and reducing latency can be found in Li et al. [72]. The conclusion of that work is that even in large networks with high churn, low-latency routing can be obtained with a minimal maintenance cost (on the order of 10 bytes per second per node) using Accordion, a lookup system that DHash can use instead of Chord. Accordion adapts its routing table size to the environment in which it is run, freeing DHash from considering this tradeoff.

The use of erasure coding can decrease maintenance costs for data at the expense of increasing lookup latency when compared to creating full replicas. Figure 4-10 showed that using erasure coding increases fetch latency: since more servers must be contacted compared to replication the probability that one of the servers will be distant in the network is greater. However, in Section 6 we observed that erasure coding decreases the cost of maintaining data and allows greater durability at the same storage overhead. An erasure code improves the durability of the system by letting the system create new copies of data faster (each fragment is much smaller than a full replica); this increases ρ. Unlike routing information, there does not appear to be a solution that provides both low latency and low maintenance traffic. The "penalty" for not using coding is large in terms of disk space, insert throughput, and durability.

DHash consistently trades performance for scalability. For example, Chord chooses higher-latency lookups over keeping more state. The design of STP similarly trades performance for scalability: if an application limits itself to running on a handful of nodes (< 100), DHash could use persistent TCP connections between all of the nodes. The danger of such an approach is that an open system will suddenly become popular. It might be possible to design a transport that smoothly transitions from N^2 TCP connections to a more scalable transport like STP as the system grows.

8.3 Future work

DHash, in its current state as deployed on PlanetLab, functions well-enough to support several research systems. If the system is to move into wider deployment, a number of engineering challenges remain to be tackled.

8.3.1 Network

Implicit in the design of Chord is the assumption that all nodes in the system can communicate with one another. In practice it is occasionally the case that among three nodes a, b, and c, a

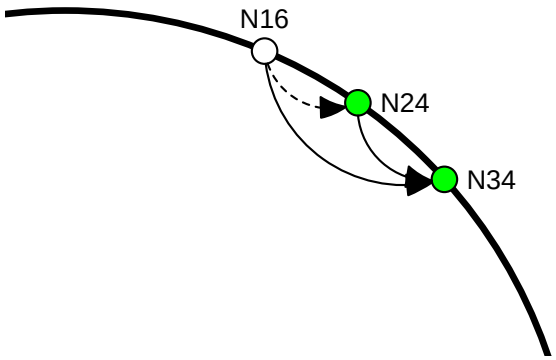


Figure 8-1: The effect of non-transitive network communication on the consistency of the Chord identifier space: actual successor pointers are shown as solid lines; ideal pointers are shown as dotted lines. In this example, node N16 cannot communicate with its ideal-successor, N24, but can communicate with other nodes (including N34). When lookups for keys in the range [16..24) arrive at N16, N16 incorrectly reports that N34 is the successor node.

can communicate with c but not b while c can communicate with both a and b; we call this non-transitive connectivity. Non-transitivity occurs on PlanetLab mainly due to the use of Internet2 nodes. These nodes can only communicate with other Internet2 hosts. However, some hosts, mainly Universities, are connected to both Internet2 and the commercial Internet; a trio of one Internet2 host, one commercial host, and a university lead to non-transitive connectivity.

When this non-transitive communication occurs, Chord does not function correctly. For instance, the consistent hashing mapping that underlies Chord is no longer well-defined: if a node mistakenly believes that its successor has failed, it will incorrectly assign ownership of its successor's portion of the key space to the following node in the key space. Figure 8-1 shows an example how non-transitive communication distorts the successor lists of three nodes. Other effects of non-transitive network communication include increased maintenance traffic (from the example, N16 is repeatedly told by N34 that its successor should be N24), and RPC timeouts (N16 must time out an RPC to N24 once each time it learns about N24's existence). The impact of non-transitivity on DHash's correctness is currently not severe: even though lookups are routed to the incorrect node they usually succeed since enough fragments of the block are still able to be located. In the example above, only one fragment is unreachable. Others have observed these same problems [43, 73].

Future versions of DHash could avoid this problem by applying lessons from RON [4]. A node (N) might remember the identity of the host that supplied the identity of each other node in its routing table (S). If a node only pass on information about nodes that it can communicate with, non-transitivity could be eliminated by performing a one-hop indirect route through S.

The problem of NATs also looms large over wide-spread DHT deployment: DHash acts as a server in numerous roles. On promising line of work involves communicating through NATs as long as a third host arranges the rendezvous [41].

8.3.2 Trust and cooperative behavior

This thesis has assumed that all nodes in the system trust one another. This assumption may not be reasonable in practice. Malicious, or simply buggy participants can disrupt the operation of DHash in several ways. A malicious node could simply drop all recursive

lookup messages that pass through it, or accept blocks for storage and then delete them. Because DHTs are intended to be open networks, an attacker could create a large number of nodes and effectively gain control over the system; this vulnerability is known as the Sybil attack [38].

The self-certifying block names used by DHash limit the damage that malicious nodes can cause to denial of service attacks. Notably, a malicious node can not alter the contents of a block without inverting SHA-1 or forging an RSA signature.

Some work has been done to limit the damage that a minority group of malicious hosts can wreak [15]. Systems such as these penalize lookup latency (by forcing iterative lookups, for instance) or erode the design goals of DHTs (by relying on central certification authorities).

An alternative is to build routing structures on top of "real world" trusted links. For example, one might build a routing table from one's IM "buddy list" or Orkut friend list. It's not clear how to route efficiently on this graph (although research into the small world effect suggests that it may be possible [67]). However, the advantage of this approach is that the system can use lightweight security mechanisms and rely on the underlying trust relationships to provide some guarantee that hosts are not malicious [100, 36].

8.3.3 Maintenance Traffic

While DHash's replication algorithms are far more efficient than the initial prototype's algorithms, recent experience with an application (UsenetDHT) that stores a large amount of data in DHash suggests that replication systems that react to failure may cause overload. Figure 6-12 (b) suggests this effect also: when a large number of nodes fail temporarily around day 150, the system sends a huge amount of data to maintain $r_L$replicas.

DHash, and any system designed around creating new replicas when the number of copies reaches a specified threshold, is *reactive*. Such systems are likely to contain feedback loops that may lead to unpredictable and destructive behavior. Rhea [108] demonstrates such behavior in the routing table maintenance algorithms of the Pastry DHT. In that system, repair activity was triggered by failures. Because repair activity involved sending traffic on the network, when failures were frequent the network became congested; this congestion was mistaken by the system for more failures which in turn increased the amount of traffic. As a result of this positive feedback loop the system was unusable. A similar feedback loop could occur as a result of data maintenance traffic.

To avoid this feedback, one could consider a system that does not react to failures, but instead creates new replicas at a constant, user specified, rate. The system maximizes availability given a bandwidth cap that it guarantees to never exceed (this cap will usually be specified as some fraction of a node's link bandwidth); this is in contrast to systems which use as much bandwidth as necessary to meet an availability specification (given explicitly or in the form of a minimum replication level). Both systems offer the same availability guarantees, namely best-effort availability.

Because the equilibrium between block creation and failure rates determines the number of replicas that the system maintains, the system is adapting to the environment (namely the failure rate) by varying the replication level. Systems that specify a number of replicas adapt by varying the bandwidth usage in an attempt (possibly unsuccessful) to maintain that replication level.

In addition to not overloading nodes, the system uses network resources well. Network links are usually priced by capacity rather than per byte sent. This means that any time a link is idle, resources are being wasted. By sending at a constant rate, this system does not let

links go idle. Additionally, although this system may send more bytes than strictly necessary, it distributes over time when those bytes must be sent rather than concentrating network usage around failure times: this better suits the properties of low-capacity network links that do not absorb bursts well. However, if network capacity was priced per byte sent, a strategy that minimized the number of bytes sent would be more appropriate.

The cost of this approach is increased disk usage. The number of replicas in a long-running system is ρ (see Section 6.2) In a stable system, this value is likely to be larger than the minimum number of replicas required to maintain availability. The philosophy of this system with respect to the tradeoff between disk and network resources is that disks can be "unfilled" but there is no way to go back in time and send bytes on a link that was left idle.

It should be possible to modify Sostenuto to take a non-reactive approach by limiting the number of repairs per unit time using a token bucket. Periodically, Sostenuto will prioritize blocks based on how many replicas are remaining (blocks with less replicas get priority) and perform as many repair actions as the bandwidth budget allows.

Bibliography

[1] *Proceedings of the 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002).

[2] *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004).

[3] *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005).

[4] Andersen, D., Balakrishnan, H., Kaashoek, M. F., and Morris, R. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Chateau Lake Louise, Banff, Canada, October 2001).

[5] Andersen, D., Balakrishnan, H., Kaashoek, M. F., and Morris, R. Experience with an evolving overlay network testbed. *ACM SIGCOMM Computer Communication Review 33*, 3 (July 2003), 13–19.

[6] Anderson, R. J. The eternity service. In *Pragocrypt 96* (1996).

[7] Bakker, A., Amade, E., Ballintijn, G., Kuz, I., Verkaik, P., van der Wijk, I., van Steen, M., and Tanenbaum., A. The Globe distribution network. In *Proc. 2000 USENIX Annual Conf. (FREENIX Track)* (San Diego, CA, June 2000), pp. 141–152.

[8] Battista, G. D., Eades, P., Tomassia, R., and Tollis, I. Annotated bibliography on graph drawing algorithms. *Computational Geometry: Theory and Applications* (1994).

[9] Bershad, Y. S. B. N., and Levy, H. M. Manageability, availability, and performance in porcupine: a highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (Kiawah Island, South Carolina, December 1999).

[10] Bhagwan, R., Tati, K., Cheng, Y.-C., Savage, S., and Voelker, G. M. Total Recall: System support for automated availability management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* [2].

[11] BitTorrent. `http://bitconjurer.org/BitTorrent/protocol.html`.

[12] Blake, C., and Rodrigues, R. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems* (May 2003).

[13] Calvert, K. L., Doar, M. B., and Zegura, E. W. Modeling Internet topology. *IEEE Communications 35*, 6 (June 1997), 160–163.

[14] Castro, M., Costa, M., and Rowstron, A. Debunking some myths about structured and unstructured overlays. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation* [3].

[15] Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. S. Security for structured peer-to-peer overlay networks. In *5th Symposium on Operating Systems Design and Implementaion (OSDI'02)* (Dec. 2002).

[16] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. Exploiting network proximity in peer-to-peer overlay networks. Tech. Rep. MSR-TR-2002-82, Microsoft Research, June 2002.

[17] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Tech. Rep. MSR-TR-2003-52, Microsoft Research, 2003.

[18] Castro, M., and Liskov, B. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20*, 4 (2002), 398–461.

[19] Cates, J. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[20] Chamberlin, D. D., Astrahan, M. M., Blasgen, M. W., Gray, J. N., King, W. F., Lindsay, B. G., Lorie, R., Mehl, J. W., Price, T. G., Putzolu, F., Selinger, P. G., Schkolnick, M., Slutz, D. R., Traiger, I. L., Wade, B. W., and Yost, R. A. A history and evaluation of system r. *Commun. ACM 24*, 10 (1981), 632–646.

[21] Chen, Y., Edler, J., Goldberg, A., Gottlieb, A., Sobti, S., and Yianilos, P. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital libraries* (Berkeley, CA, Aug. 1999), pp. 28–37.

[22] Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., and Tuecke, S. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications 23* (2001), 187–200. (based on conference publication from Proceedings of NetStore Conference 1999).

[23] Chiu, D.-M., and Jain, R. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems 17* (1989), 1–14.

[24] Chord website. `http://pdos.lcs.mit.edu/chord/`.

[25] Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. Planetlab: an overlay testbed for broad-coverage services. In *SIGCOMM Computer Communications Review* (August 2003), pp. 3–12.

[26] Clarke, I. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[27] Cohen, B. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems* (Berkeley, CA, 2003).

[28] Costa, M., Castro, M., Rowstron, A., and Key, P. PIC: Practical Internet coordinates for distance estimation. In *International Conference on Distributed Systems* (Tokyo, Japan, March 2004).

[29] Cox, L. P., and Noble, B. D. Pastiche: making backup cheap and easy. In *OSDI 2002* (Dec. 2002).

[30] Cox, R., and Dabek, F. Learning Euclidean coordinates for Internet hosts. `http://pdos.lcs.mit.edu/~rsc/6867.pdf`, December 2002.

[31] Cox, R., Dabek, F., Kaashoek, F., Li, J., and Morris, R. Practical, distributed network coordinates. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)* (Cambridge, Massachusetts, November 2003).

[32] Dabek, F. A cooperative file system. Master's thesis, Massachusetts Institute of Technology, September 2001.

[33] Dabek, F., Cox, R., Kaashoek, F., and Morris, R. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM* (August 2004).

[34] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001), pp. 202–205.

[35] Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M. F., and Morris, R. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)* (San Francisco, California, March 2004).

[36] Danezis, G., Lesniewski-Laas, C., Kaashoek, M. F., and Anderson, R. Sybil-resistant DHT routing. In *Proceedings of the 10th European Symposium On Research In Computer Security (ESORICS)* (2005).

[37] de Launois, C., Uhlig, S., and Bonaventure, O. A stable and distributed network coordinate system. Tech. rep., Universite catholique de Louvain, Belgium, November 2004.

[38] Douceur, J. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems* [1].

[39] Drake, A. W. *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967.

[40] Fonseca, R., Ratnasamy, S., Zhao, J., Ee, C. T., Culler, D., Shenker, S., and Stoica, I. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation* [3].

[41] Ford, B., Srisuresh, P., and Kegel, D. Peer-to-peer communication across network address translators. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX '05)* (Anaheim, California, April 2005).

[42] Francis, P., Jamin, S., Jin, C., Jin, Y., Raz, D., Shavitt, Y., and Zhang, L. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Transactions on Networking* (Oct. 2001).

[43] Freedman, M. J., Lakshminarayanan, K., Rhea, S., and Stoica, I. Non-transitive connectivity and dhts. In *Proceedings of the USENIX WORLDS* (December 2005).

[44] Fu, K., Kaashoek, M. F., and Mazieres, D. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)* (San Diego, California, October 2000). Extended version in ACM Trans. on Computer Systems.

[45] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Symposium on Operating Systems Principles* (2003).

[46] Gil, T., Li, J., Kaashoek, F., and Morris, R. Peer-to-peer simulator, 2003. http://pdos.lcs.mit.edu/p2psim.

[47] Giles, C. L., Bollacker, K. D., and Lawrence, S. Citeseer: an automatic citation indexing system. In *DL '98: Proceedings of the third ACM conference on Digital libraries* (1998), pp. 89–98.

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

[49] Gribble, S. D., Brewer, E. A., Hellerstein, J. M., and Culler, D. Scalable, distributed data structures for Internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)* (October 2000).

[50] Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S., and Stoica, I. The impact of DHT routing geometry on resilience and proximity. In *Proc. of SIGCOMM 2003* (August 2003).

[51] Gummadi, K. P., Saroiu, S., and Gribble, S. D. King: Estimating latency between arbitrary Internet end hosts. In *Proc. of SIGCOMM IMW 2002* (November 2002), pp. 5–18.

[52] Gupta, A., Krauthgamer, R., and Lee, J. R. Bounded geometries, fractals, and low-distortion embeddings. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)* (2003), p. 534.

[53] Gupta, A., Liskov, B., and Rodrigues, R. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st NSDI* (Mar. 2004).

[54] Gupta, I., Birman, K., Linga, P., Demers, A., and van Renesse, R. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd IPTPS* (Feb. 2003).

[55] Hand, S., and Roscoe, T. Mnemosyne: Peer-to-peer steganographic storage. In *1st International Workshop on Peer-to-Peer Systems (IPTPS'02)* (Mar. 2001).

[56] Harvey, N., Jones, M. B., Saroiu, S., Theimer, M., and Wolman, A. Skipnet: A scalable overlay network with practical locality properties. In *In proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)* (Seattle, WA, March 2003).

[57] Harvey, N. J. A., Jones, M. B., Theimer, M., and Wolman, A. Efficient recovery from organizational disconnects in skipnet. In *IPTPS* (2003), pp. 183–196.

[58] Hooke, R. De potentia restitutiva (lecture), 1679.

[59] Hoppe, H. *Surface reconstruction from unorganized points*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1994.

[60] Iyer, S., Rowstron, A., and Druschel, P. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*. (July 2002).

[61] Jacobson, V. Congestion avoidance and control. In *Proc. ACM SIGCOMM* (Aug. 1988), pp. 314–329.

[62] Kaashoek, M. F., and Karger, D. R. Koorde: A simple degree-optimal hash table. In *Proceedings of the 2nd IPTPS* (2003).

[63] Karger, D. R., and Ruhl, M. Finding nearest neighbors in growth-restricted metrics. In *ACM Symposium on Theory of Computing (STOC '02)* (Montreal, May 2002).

[64] Karp, B., Ratnasamy, S., Rhea, S., and Shenker, S. Spurring adoption of DHTs with OpenHash, a public DHT service. In *Proceedings of the 3rd IPTPS* (Feb. 2004).

[65] KaZaA media dekstop. `http://www.kazaa.com/`.

[66] Kleinbeg, J., Slivkins, A., and Wexler, T. Triangulation and embedding using small sets of beacons. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)* (2004), pp. 444–453.

[67] Kleinberg, J. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the 32nd STOC* (2000).

[68] Kohler, E., Handley, M., and Floyd, S. Datagram congestion control protocol (DCCP). IETF Internet draft, 2005.

[69] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, November 2000), pp. 190–201.

[70] Lee, E. K., and Thekkath, C. A. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1996), pp. 84–92.

[71] Li, J., Jannotti, J., De Couto, D., Karger, D., and Morris, R. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking* (Boston, Massachusetts, August 2000), pp. 120–130.

[72] Li, J., Stribling, J., Morris, R., and Kaashoek, M. F. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)* (Boston, Massachusetts, May 2005).

[73] Li, J., Stribling, J., Morris, R., Kaashoek, M. F., and Gil, T. M. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of the 24th Infocom* (Miami, FL, March 2005).

[74] Linial, N., London, E., and Rabinovich, Y. The geometry of graphs and some of its algorithmic applications. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS'94)* (November 1994), pp. 577–591.

[75] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M. Replication in the Harp file system. In *Symposium on Operating Systems Principles* (1991), pp. 226–38.

[76] Litwin, W., Neimat, M.-A., and Schneider, D. A. LH* a scalable, distributed data structure. *ACM Transactions on Database Systens 21*, 4 (1996), 480–525.

[77] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of OSDI* (San Francisco, CA, 2004).

[78] Malkhi, D., Naor, M., and Ratajczak, D. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of Principles of Distributed Computing (PODC 2002)* (July 2002).

[79] Matousek, J. Open problems. In *Workshop on Discrete Metric Spaces and their Algorithmic Applications* (Updated September, 2005).

[80] Maymounkov, P., and Mazieres, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002). full version in the Springer Verlag proceedings, `http://kademlia.scs.cs.nyu.edu/pubs.html`.

[81] Mazieres, D. A toolkit for user-level file systems. In *USENIX Technical Conference* (2001).

[82] Medina, A., Lakhina, A., Matta, I., and Byers., J. BRITE: An approach to universal topology generation. In *Proceedings of the Ninth IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'01)* (2001).

[83] Merkle, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology - Crypto '87* (Berlin, 1987), C. Pomerance, Ed., Springer-Verlag, pp. 369–378. Lecture Notes in Computer Science Volume 293.

[84] Mockapetris, P., and Dunlap, K. J. Development of the Domain Name System. In *Proc. ACM SIGCOMM* (Stanford, CA, 1988), pp. 123–133.

[85] Mogul, J. C. Efficient use of workstations for passive monitoring of local area networks. Research Report 90/5, Digital Western Research Laboratory, July 1990.

[86] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th OSDI* (Dec. 2002).

[87] Napster. `http://www.napster.com/`.

[88] Ng, E. GNP software, 2003. `http://www-2.cs.cmu.edu/~eugeneng/research/gnp/software.html`.

[89] Ng, T. E., and Zhang, H. A network positioning system for the Internet. In *Proceedings of the USENIX Conference* (June 2004).

[90] Ng, T. S. E., and Zhang, H. Predicting Internet network distance with coordinates-based approaches. In *Proceedings of IEEE Infocom* (2002), pp. 170–179.

[91] Ong, L., and Yoakum, J. An introduction to SCTP. Rfc, 2002.

[92] OpenDHT project. `http://opendht.org`.

[93] Oppenheimer, D., Albrecht, J., Patterson, D., and Vahdat, A. Distributed resource discovery on planetlab with sword. In *Proceedings of the First workshop on real large distributed systems (WORLDS'04)* (2004).

[94] Padmanabhan, V., and Subramanian, L. An investigation of geographic mapping techniques for Internet hosts. In *Proceedings of the ACM SIGCOMM* (San Diego, Aug. 2001), pp. 173–185.

[95] Park, K. S., and Pai, V. CoMon: A monitoring infrastructure for planetlab. `http://comon.cs.princeton.edu/`.

[96] Patterson, D., Gibson, G., and Katz, R. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June 1988).

[97] Pias, M., Crowcroft, J., Wilbur, S., Harris, T., and Bhatti, S. Lighthouses for scalable distributed location. In *IPTPS* (2003).

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

[99] Plaxton, C., Rajaraman, R., and Richa, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA* (Newport, Rhode Island, June 1997), pp. 311–320.

[100] Popescu, B. C., Crispo, B., and Tanenbaum, A. S. Safe and Private Data Sharing with Turtle: Friends Team-Up and Beat the System. In *Proceedings of the 12th Cambridge International Workshop on Security Protocols* (April 2004).

[101] Prityantha, N. *The Cricket Indoor Location System*. PhD thesis, MIT, June 2005.

[102] Priyantha, N., Balakrishnan, H., Demaine, E., and Teller, S. Anchor-free distributed localization in sensor networks. Tech. Rep. TR-892, MIT LCS, Apr. 2003.

[103] Priyantha, N., Chakraborty, A., and Balakrishnan, H. The Cricket Location-Support System. In *Proc. 6th ACM MOBICOM Conf.* (Boston, MA, Aug. 2000).

[104] Rabin, M. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36*, 2 (1989), 335–348.

[105] Rao, A., Ratnasamy, S., Papadimitriou, C., Shenker, S., and Stoica, I. Geographic routing without location information. In *ACM MobiCom Conference* (Sept. 2003), pp. 96 – 108.

[106] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001), pp. 161–172.

[107] Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., and Kubiatowicz, J. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)* (Apr. 2003).

[108] Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference* (June 2004).

[109] Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., , and Yu, H. OpenDHT: A public DHT service and its uses. In *Proceedings of ACM SIGCOMM 2005* (August 2003).

[110] Rinaldi, R., and Waldvogel, M. Routing and data location in overlay peer-to-peer networks. Research Report RZ–3433, IBM, July 2002.

[111] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-s cale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001).

[112] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001).

[113] Saito, Y., Frlund, S., Veitch, A., Merchant, A., and Spence, S. Fab: building distributed enterprise disk arrays from commodity components. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, 2004), ACM Press, pp. 48–58.

[114] Savarese, C., Rabaey, J. M., and Beutel, J. Locationing in distributed ad-hoc wireless sensor networks. In *ICASSP* (May 2001), pp. 2037–2040.

[115] Shavitt, Y., and Tankel, T. Big-bang simulation for embedding network distances in Euclidean space. In *Proc. of IEEE Infocom* (April 2003).

[116] Shavitt, Y., and Tankel, T. On the curvature of the Internet and its usage for overlay construction and distance estimation. In *Proc. of IEEE Infocom* (April 2004).

[117] Sidiropoulos, A. *Approximation Algorithms for Low-Distortion Embeddings into Low-Dimensional Spaces*. PhD thesis, MIT, May 2005.

[118] Sit, E., Dabek, F., and Robertson, J. UsenetDHT: A low overhead usenet server. In *Proceedings of the 3rd IPTPS* (Feb. 2004).

[119] Slivkins, A. Distributed approaches to triangulation and embedding. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'05)* (2005), pp. 640–649.

[120] Stevens, W. R. RFC2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Tech. rep., Internet Assigned Numbers Authority, 1997.

[121] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM 2002* (August 2002).

[122] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM* (San Diego, Aug. 2001). An extended version appears in ACM/IEEE Trans. on Networking.

[123] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking* (2002), 149–160.

[124] Stribling, J. All-pairs-ping trace of PlanetLab, 2004. `http://pdos.lcs.mit.edu/~strib/pl\_app`.

[125] Stribling, J. Overcite: A cooperative digital research library. Master's thesis, Massachusetts Institute of Technology, September 2005.

[126] Tam, P. *A Physicist's Guide to Mathematica*. Elsevier, 1997.

[127] Tang, L., and Crovella, M. Virtual landmarks for the Internet. In *Internet Measurement Conference* (Miami Beach, FL, October 2003), pp. 143 – 152.

[128] Thekkath, C. A., Mann, T., and Lee, E. K. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles* (1997), pp. 224–237.

[129] van Renesse, R., and Schneider, F. B. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operationg Systems Design and Implementation* (Dec. 2004).

[130] van Steen, M., Hauck, F., Ballintijn, G., and Tanenbaum, A. Algorithmic design of the Globe wide-area location service. *The Computer Journal 41*, 5 (1998), 297–310.

[131] Waldvogel, M., and Rinaldi, R. Efficient topology-aware overlay network. In *Hotnets-I* (2002).

[132] Walfish, M., Balakrishnan, H., and Shenker, S. Untangling the web from DNS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* [2].

[133] Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., and Shenker, S. Middleboxes no longer considered harmful. In *Proc. USENIX OSDI* (San Francisco, CA, December 2004).

[134] Wang, L., Pai, V., and Peterson, L. The Effectiveness of Request Redirecion on CDN Robustness. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA USA, December 2002).

[135] Weatherspoon, H., and Kubiatowicz, J. D. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems* [1].

[136] Weinberg, S. Learning effectiveness: The impact of response time (abstract only). In *Proceedings of the joint conference on Easier and more productive use of computer systems. (Part - II)* (New York, NY, USA, 1981), ACM Press, p. 140.

[137] Wong, B., Slivkins, A., and Sirer, E. G. Meridian: a lightweight network location service without virtual coordinates. In *Proceedings of the SIGCOMM* (August 2005).

[138] Zhao, B., Kubiatowicz, J., and Joseph, A. Tapestry: An infrastructure for fault-tolerant wide-area locat ion and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr. 2001.

[139] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (Jan. 2004).

Index