OpenDHT: A Public DHT Service

by

Sean Christopher Rhea

B.S. (University of Texas, Austin) 1998

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Fall 2005

The dissertation of Sean Christopher Rhea is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2005

OpenDHT: A Public DHT Service

Abstract

OpenDHT: A Public DHT Service

by

Sean Christopher Rhea
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Kubiatowicz, Chair

The distributed hash table, or DHT, is a distributed system that provides a traditional hash table's simple put/get interface using a peer-to-peer overlay network. DHTs deliver incremental scalability in the number of nodes, high availability, and low latency.

We present the Bamboo DHT and the OpenDHT public DHT service. Bamboo supports low-latency under very high churn rates; with session times as short as six minutes, a 1000-node Bamboo network on ModelNet is still able to average around one half second per get operation. Bamboo also supports reliable, high-performance storage on a 200–300 node PlanetLab deployment. It provides very high availability as measured over months, and it maintains very low get latencies despite the presence of arbitrarily slow nodes. Furthermore, Bamboo is resilient to non-transitivity in the underlying network, a requirement for long-term use on the Internet.

OpenDHT is a public DHT service designed to ease the deployment and maintenance of DHT-based applications. By providing an existing deployment with a simple put/get interface over RPC, OpenDHT allows the construction of DHT applications in tens of lines of code. OpenDHT provides a simple, secure put/get/remove interface, and it also supports more sophisticated features such as anycast, multicast, and range search using client-side libraries. Furthermore, OpenDHT guarantees a fair share of storage to each client in the system, while preventing any client from going long periods without being able to perform any puts at all. At the time of this writing, OpenDHT has been running as a public service on PlanetLab for over 16 months.

Professor John Kubiatowicz
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgments

John Kubiatowicz, my advisor, first got me excited about peer-to-peer systems. The first talk I saw him give was the OceanStore pitch, and I caught the fever immediately. This thesis is the result of that six-year obsession. Kubi is also a master of pedagogy, and every paper and talk I have written has ended up remarkably more clear as a result of his advise.

Scott Shenker convinced me that a public DHT service made sense, despite my stubborn refusal to see it at the time. His insistence on simplicity as a path to robustness has shaped the way I think about systems ever since, and I am indebted to him for it.

Ion Stoica has given me excellent advice at several key points in my time at Berkeley, and his imprint can been seen in both Bamboo and OpenDHT. His insights on the FST algorithm, in particular, were vital.

Eric Brewer and Joe Hellerstein, in teaching the graduate systems course my first semester at Berkeley, convinced me to change my research focus from programming languages to systems. I've since worked for both of them—for Eric while at Inktomi and for Joe while at Intel—and have benefited from their advise throughout.

John Chuang graciously agreed to be on my qualifying exam committee and to be a reader for my thesis, despite being swamped by such requests from the CS department.

At Intel, Brad Karp, Sylvia Ratnasamy, and Timothy Roscoe have been excellent mentors and co-authors of mine. Brad and Sylvia helped Scott turn me on to the OpenDHT idea, and I've enjoyed working with them ever since. Mothy has a systems intuition shaped by experiences from multimedia operating systems to telephone networks, and I always appreciate his diverse perspective.

Outside of professors and mentors, I've been blessed with a wonderful group of additional co-authors over the years. Dennis Geels contributed much of the work that has ended up in Chapter 3 of this thesis, writing the benchmarking scripts, helping me understand what was wrong with the other DHT implementations, and being the sounding board against which I tried out my early ideas for Bamboo.

The other core members of the Pond crew, Patrick Eaton and Hakim Weatherspoon, have also been vital sounding boards for ideas, and through their early adoption of Bamboo in Pond, helped me debug and refine my code as well.

The version of ReDiR presented in Chapter 5, a significant improvement over our earlier work [KRRS04], is the work of Brighten Godfrey. Mike Freedman and Karthik Lakshminarayanan are responsible for the insights into Chord and Kademlia that appear in Chapter 6. Byung-Gon Chun, my coauthor on the paper that became Chapter 7, built the testing framework used there and contributed several important insights.

I was first motivated to apply to graduate school by three professors at the University of Texas—Adnan Aziz, Jeff Chase, and Vijay Garg—all three of whom later helped in getting me into Berkeley by writing my letters.

Likewise, Eric Brewer, John Kubiatowicz, Scott Shenker, Ion Stoica, Remzi Arpaci-Dusseau, and Jody Lewen all wrote much-appreciated letters of recommendation for my job search at the end of graduate school.

Several of members of the EECS department administration—Mary Byrnes, Ruth Gjerde, Peggy Lau, and La Shana Porlaris—have helped me navigate the university's ample red tape.

I have been financially supported throughout this work by the National Science Foundation and IBM through research grants and fellowships, and by Inktomi and Intel through summer internships.

My brother, friends, and roommates have been wonderfully supporting throughout my graduate career, listening to my frustrations and understanding when I had to work late rather than socialize. Likewise, my dog has endured far to little attention at times.

Finally, I must give much credit and thanks to my parents, who kick-started this whole process when they enrolled me in Logo programming classes at the age of eight and later bought my brother and I an Apple II computer of our own. I wonder if they knew then what they were getting me into.

Chapter 1

Introduction

Large-scale distributed systems are notoriously difficult to design, implement, deploy, and debug. Consequently, there is a long history of research that aims to ease the construction of such systems by providing simple primitives on which more sophisticated functionality can be built. Examples include remote procedure call (RPC) [BN84], the domain name system (DNS) [MD88]; fault-tolerant, replicated state machines [CRL03]; causal, atomic multicast [BSS91]; and epidemic communication primitives [DGH+87].

A more recent addition to this list is the distributed hash table, or DHT [RD01, RFH+01, SMK+01, ZHS+04]. A hash table is a data structure that provides two main functions: *put* takes a key and value and stores them in the table, and *get* takes a key and returns the value (if any) previously put under that key. A distributed hash table provides this same interface, but partitions the key space across a set of cooperating peers to provide incremental scalability, and replicates each key-value pair for high availability.

Figure 1.1 shows an example DHT. Seven nodes participate in the system, and the key space is divided between them. An overlay network connects the nodes. When the node in the lower left corner puts a key-value pair into the DHT, the put request is routed to the node responsible for the key in question, and that node stores the pair. When the node on the lower right later tries to get all values with that key, the get request is routed to the same node, and it returns the value previously put.

Without going into detail, we review the basic benefits of DHTs. First, they are scalable; additional capacity can be added to the system by simply adding more nodes. Second, they are fault-tolerant; although not shown in the picture, DHTs generally store each value on several nodes, so that no value will be lost with the failure of any one node. A related point is that DHTs are

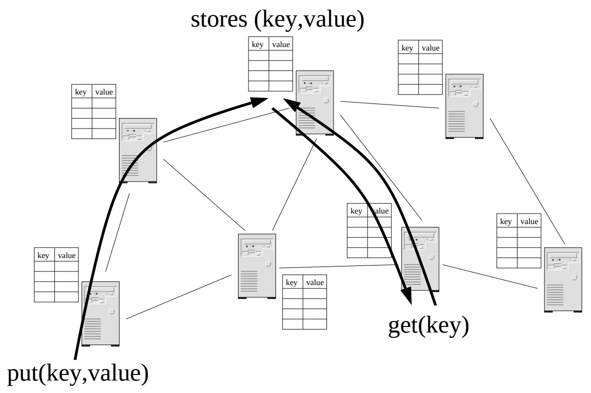


Figure 1.1: *A distributed hash table.*

completely decentralized; there is no one point of control, and hence no one node that will disable the system by its failure. Third, DHTs can be built up entirely on existing resources; there is no need for expensive, managed infrastructure on which to host the system.

In practice, there are a number of challenges to building such a system, but before we move forward from this simple description, we first describe two DHT applications, in order to give a concrete notion of the utility of DHTs.

1.1 Two Example DHT Applications

Two DHT applications—the Coral content distribution network [FFM04] and the FeedTree cooperative news service [SMPD05]—have recently gained some popularity. Although we did not build these applications ourselves, they nicely illustrate the utility of DHTs. In this section we describe both applications at a high level. Our intent here is purely pedagogical; we will omit or modify details as necessary for clarity of presentation. For a complete and accurate description of either application, the reader is referred to the publications cited above.

1.1.1 Coral

The Coral content distribution network (CDN) allows a group of web sites to protect themselves against flash crowds by cooperatively serving each others' content. Figure 1.2 illustrates the Coral system. When Client 1 fetches a web page from server foo.com, Coral intercepts request, routing it through server bar.com. When foo.com responds, bar.com caches a copy of the response. When Client 2 later fetches the same web page, Coral routes the request through bar.com,

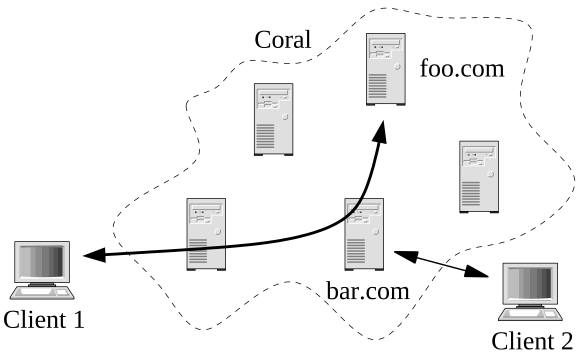


Figure 1.2: *The Coral content distribution network.*

which returns the web page directly, without contacting foo.com a second time. In this way, Coral shifts load for foo.com onto its peers; under light usage, this technique does not significantly help performance, but when a single server experiences a surge in popularity, its peers help it to satisfy the demand.

To reduce the load on a web server, Coral must ensure that subsequent requests for that server's web pages are routed through other sites in the system that already contain cached copies of those pages. It achieves this feat through the partitioning performed by a DHT; for any given resource, it uses this partitioning to select a set of servers that will act as caches for that resource. Instead of using a DHT, however, Coral could of course use some centralized index of resources, or even a centralized cache. We thus review why a DHT is a particularly good fit for Coral.

One major advantage of using a DHT in Coral is that the system contains no natural point of centralization. The web sites that make up the system are each small and relatively low bandwidth; otherwise, they could handle their own flash crowds and would have little motivation to participate. While the sites could band together to pay for high-bandwidth, centralized service from some third party, Coral instead allows them to utilize the bandwidth they already have to accomplish the same goal.

Another benefit of using a DHT in Coral is that because there is no one point of centralization, any site can opt out of the system at any time. While doing so reduces the effective bandwidth in the system, it does so only in proportion to the amount of bandwidth that the departing site was providing. Likewise, when a new site joins, the net bandwidth of the system increases by the bandwidth available from that site. For this reason, we say that the system is *incrementally scalable*.

While incremental scalability is attractive in its own right, it also provides a secondary

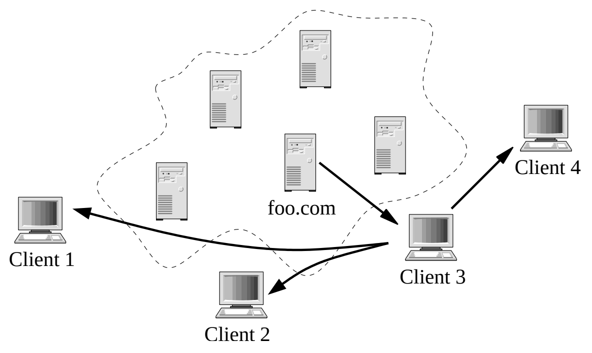


Figure 1.3: *Cooperative RSS dissemination with FeedTree.*

advantage to the system in that it creates a so-called *network effect*: the more sites that join the system, the larger a flash crowd the system as a whole can handle, and the more attractive it is for other nodes to join.

The three advantages Coral achieves by using a DHT—decentralization, incremental scalability, and the network effect—are also realized by FeedTree, as we describe next.

1.1.2 FeedTree

FeedTree is a system for the cooperative dissemination of news. Because it is based on the Really Simple Syndication (RSS) protocol [rss], we describe that first.

The RSS protocol encapsulates diverse news feeds in a common XML-based format. Each feed usually includes an XML item containing a story title and sometimes a summary or abstract. Client programs aggregate multiple feeds into a common user interface. Like the web, RSS uses a pull model, where clients periodically poll the news sources in which they are interested to check whether their feeds have been updated. Unlike the web, however, this polling is usually automated (as opposed to click-driven). Since this automated polling can severely tax a news source, most sources limit each client to a minimum polling period (usually 30 minutes).

As shown in Figure 1.3, FeedTree [SMPD05] uses a DHT-based multicast system to push updated news feeds to clients more quickly. For each news feed f (e.g., *foo.com* in the figure) in which it is interested, a FeedTree client a registers its interest with a DHT (essentially doing a $put(f,a)$); when a client discovers a update to a feed f, it multicasts the update to all interested clients, which it discovers using the DHT (essentially $get(f)$).

A major advantage of FeedTree is that it can be deployed today; there is no need for cooperation with news sites. At the same time, however, without their cooperation there are no natural points of centralization in the system, and for this reason a DHT is attractive. Also, as with Coral, FeedTree benefits from a network effect; the more clients that join the system to monitor a particular news feed, the shorter the aggregate polling period becomes, and the quicker each client sees each news item. Finally, as in Coral, FeedTree clients utilize bandwidth and computational resources they already have to achieve improved performance; there is no need to pay for this additional service.

An interesting feature of the FeedTree design is that it uses a single DHT for all news feeds, as opposed to using a different DHT for each feed. One difficulty with DHTs is called the *bootstrap problem*; in order to join the DHT, a node must know of one other node that has already joined. By using a single DHT for all feeds, FeedTree needs only solve the bootstrap problem once, amortizing that cost across all feeds.

1.1.3 Discussion

We chose to discuss the two applications above because they are in some senses perfect candidate applications for a DHT. They have no natural point of centralization, they are able to utilize resources they already have at hand, and the DHT's scalability allows them to take advantage of network effects. There are many other applications that benefit from the use of DHTs, and we will describe several more in the course of this work. There are, of course, other applications for which a DHT is not a good match. In Sections 2.2 and 2.3 we present a more thorough discussion of the strengths and limitations of DHTs.

1.2 OpenDHT: The DHT as a Service

The FeedTree application above nicely illustrates a particular point; for clients interested in some new news feed that is not yet being multicast by the system, the existence of an up-and-running DHT to which they are already connected provides an advantage. Rather than trying to discover another node interested in the same news feed in order to bootstrap itself into a feed-specific DHT, a client simply does a put to join the multicast group for that feed, amortizing the cost of solving the bootstrap problem across all feeds.

Following this reasoning, if multiple distinct *feeds* can share the same DHT, it is natural to ask under what conditions multiple distinct *applications* can share the same DHT. In other words, in the same way that we amortize the bootstrap problem across feeds in FeedTree, can we also amortize it across applications through the use of a shared DHT?

Taking this reasoning one step further, we have also explored whether many—if not most—DHT applications can benefit from sharing a *single* DHT deployment. To test this hypothesis, we have built and deployed a system we call OpenDHT. OpenDHT has been running continuously on approximately 200–300 widely dispersed Internet hosts since April 2004. Each of these hosts runs an instance of the Bamboo DHT, the DHT we built, and accepts put and get requests from clients outside the system over RPC.

Because OpenDHT operates on a set of infrastructure nodes, no application need concern itself with DHT deployment, but neither can it run application-specific code on these infrastructure nodes. This is quite different than most other uses of DHTs, in which the DHT code is invoked as a library on each of the nodes running the application. The library approach is very flexible, as one can put application-specific functionality on each of the DHT nodes, but each application must deploy its own DHT. The service approach adopted by OpenDHT offers the opposite tradeoff: less flexibility in return for less deployment burden. OpenDHT provides a home for applications more suited to this compromise.

Our early experience with OpenDHT indicates that a single, shared DHT deployment is in fact broadly useful. In Chapter 5 we further describe our experience, including building applications of our own and supporting those built by others.

1.3 Contributions

We make several contributions in this work.

1.3.1 Lookup in DHTs

The foremost function of a DHT is to partition a key space across a set of nodes. To allow clients access to this partitioning, the DHT allows a client to *lookup* the node to which any key is mapped; for brevity, we call the challenge of providing this functionality the lookup problem. In itself, the problem is simple, and most DHTs handle it in a straightforward way. For example, in one approach, each node is assigned a key at random, and each key is mapped to the node to whose key it is numerically closest.

The lookup problem becomes more interesting when new nodes join the system or existing nodes fail, and the DHT must re-partition the key space dynamically. We tested several early DHT implementations under a continuous process of arrivals and failures, and seeing that none of them performed as well as we expected, we built a new DHT called Bamboo to experiment with doing better. We made several important discoveries.

Node failure, in particular, is difficult. All DHT algorithms specify how failures should be handled, but since DHTs run on the wider Internet, it is often difficult in an implementation to quickly distinguish the failure of a node from a failure or congestion event on the path to that node. In our work, we have shown that two basic techniques are needed to surmount this problem, and we demonstrate the importance of these two techniques with comparisons to other DHT implementations.

First, a DHT should route around suspected failures quickly, in much less time than is needed to confirm that they are actual failures. The overlay networks built by DHTs have many redundant paths between two nodes, and when a primary path appears faulty, it is better to route quickly through some secondary path than to wait for the primary one to recover.

Second, DHT nodes should not recover from the failure of their neighbors in the overlay *reactively*, but *periodically*. Often times a suspected failure is in fact only a period of congestion in the network, and in reacting directly to that suspected failure by trying to find a replacement neighbor, a node runs the risk of further increasing the congestion that led to it. In the worst case, this reaction can lead to a positive feedback cycle in which a node overloads some network path, partitioning the overlay. In contrast, by recovering periodically, a DHT node decouples the rate of its own recovery traffic from the congestion it experiences, preventing such positive feedback cycles.

Third, the process of new nodes joining the network presents its own set of problems. When a new node joins, the network must re-partition the key space to give that node a share, and the new node must find suitable neighbors within the overlay network. For performance reasons, most DHT algorithms endeavor to choose some of a node's neighbors to be nearby in network latency, and the algorithms to accomplish this task are often complicated and difficult to implement. In our work we demonstrate that much of this complexity is unnecessary, that simpler methods based on random sampling do just as well for the same bandwidth cost.

Our contributions concerning lookup in DHTs appear in Chapter 3.

1.3.2 Storage in DHTs

While the lookup interface is the most general one offered by DHTs, many client applications prefer the higher-level put/get interface provided by traditional hash tables. In a simple implementation of this interface, to put a value into the DHT, a node sends a message containing the given key and value to the node discovered by looking up the key. That node then stores the key and value in a hash table in its local memory or disk. To perform a get, a message containing the key is sent to the node discovered by looking up the key; that node then does a get against its local hash table and sends back any values it finds.

Another high-level interface that is often built above lookup is called Decentralized Object Location and Routing, or DOLR [ZHS$^+$04]. In this interface, clients inform the DHT as to what objects they are storing; other clients can then query the DHT to find clients storing objects in which they are interested. DOLR is usually implemented in a manner similar to put/get; essentially, the DHT just stores pointers to objects rather than the objects themselves.

In implementing either the put/get or DOLR interface over DHT lookup, the main additional functionality that is needed is the storage of values or pointers under keys. We will thus refer to this general problem as the *storage* problem.

As with lookup, the primary challenges in the storage problem are the failure of existing nodes and the arrival of new ones. To prevent data loss due to node failure, the DHT must store data redundantly across multiple nodes. When nodes fail, this redundancy must be restored. While we were not the first to propose this idea, we developed one of the first efficient mechanisms for implementing it.

Our contributions concerning the storage problem appear in Chapter 4.

1.3.3 OpenDHT: A Public DHT Service

As noted above, many applications make such generic use of a DHT that it becomes attractive to share a single DHT deployment between them. Along these lines, we have developed and deployed OpenDHT, a public DHT service running on the PlanetLab testbed [B$^+$04] for the last 16 months. OpenDHT is shared both among applications and among clients, and each type of sharing raises a new design problem.

An Interface for a shared DHT For a DHT to be shared effectively by many different applications, its interface must balance the conflicting goals of generality and ease-of-use. Generality is necessary to meet the needs of a broad spectrum of applications, but the interface should also be easy for simple clients to use.

The lookup interface described above, while clearly quite general, is troublesome in a shared service. In particular, the strength of the lookup interface is the application-specific code that is installed in the DHT. For example, when implementing put/get on top of lookup, functionality is added to each DHT node to handle put and get messages, manage data redundancy, etc. Distributing the code for arbitrary applications to all nodes in a DHT, and running that code securely such that applications do not interfere with each other, is a challenging problem.

In contrast, the put/get interface is less flexible, allowing no access to application-specific code. This lack of flexibility limits the spectrum of applications it can support, but it frees the DHT from dealing with the vagaries of application-specific code. In the design of OpenDHT, we place a high premium on simplicity. We want an infrastructure that is simple to operate, and a service that simple clients can use. Thus the storage model, with its simple put/get interface, seems most appropriate.

To get around the limited functionality of the put/get interface, we use a novel client library, Recursive Distributed Rendezvous (ReDiR), which we describe in detail in Section 5.2.2. ReDiR, in conjunction with OpenDHT, provides the equivalent of a lookup interface for any arbitrary set of machines (inside or outside OpenDHT itself). Thus clients using ReDiR achieve the flexibility of the lookup interface, albeit with a small loss of efficiency (which we describe later).

Storage allocation in a shared DHT The second type of sharing we consider is sharing between mutually untrusting clients. In offering a put/get interface, OpenDHT is essentially a public storage facility. As observed in [RH03, BMP03], if such a system offers the persistent storage semantics typical of traditional file systems, the system will eventually fill up with orphaned data. Garbage collection of this unwanted data seems difficult to do efficiently.

OpenDHT must thus carefully manage the allocation of its storage resources between clients. While ample prior work has investigated bandwidth and CPU allocation in shared settings, storage allocation has been studied less thoroughly. In particular, there is a delicate tradeoff between fairness and flexibility: the system shouldn't unnecessarily restrict the behavior of clients by imposing arbitrary and strict quotas, but it should also ensure that all clients have access to their fair share of service.

Experiences with a shared DHT In addition the technical contributions above, we have also gained a good deal of experience particular to running a shared DHT deployment over the last 16 months. Many of the design decisions we initially felt would be most important have turned out to matter much less than others we did not expect. Simplicity and ease-of-use, for example, have shown to be of paramount importance for adoption.

Our experiences designing, deploying, and running OpenDHT are described in Chapter 5.

1.3.4 DHT Practicalities

Our final two contributions in this work concern how to deal with two realities of distributed systems rarely explored by prior work: the non-transitivity of Internet connectivity, and the problem of correct, but arbitrarily slow nodes. We cover each in a short chapter of its own.

Non-transitive connectivity A universal, but unstated, assumption of all DHT algorithms of which we are aware is transitivity of connectivity in the public Internet. In other words, if node A can contact node B, and node B can in turn contact node C, then it is always assumed to be the case that node A can also contact node C. In reality, it is well known that the Internet does not exhibit this property; while it is true in general that any two hosts can communicate with each other eventually,[1] it is also the case that failures and misconfigurations of the Internet's routing infrastructure can lead to periods of hours or longer where the transitivity of connectivity is violated.

As one might expect, overcoming the violation of this fundamental assumption of DHT designs requires modification of many parts of the system, from the lookup layer up through the storage layer. Nevertheless, the problems are not insurmountable; although we routinely see violations of transitive connectivity in our PlanetLab deployment, OpenDHT is nonetheless able to successfully perform puts and gets during these periods. We describe our work on this problem in Chapter 6.

Arbitrarily slow nodes A persistent problem with our OpenDHT deployment has been that the distribution of the latencies of get operations has a very long tail. As one might expect, this tail is caused by a few, arbitrarily slow PlanetLab nodes. We have observed disk reads that take tens of seconds, computations that take hundreds of times longer to perform at some times than others, and

[1]Note that we are only talking here about hosts on the *public* Internet; of course we expect hosts behind network address translators (NATs) or firewalls to exhibit some degree of permanent non-transitive connectivity.

internode ping times well over a second. Furthermore, the set of slow nodes is not constant over time, so we cannot very well “cherry pick” a set of good nodes on which to run OpenDHT.

While it is tempting to blame OpenDHT’s performance problems on PlanetLab, anecdotal evidence suggests that the problems we have observed with PlanetLab are common in other large-scale systems. Instead, it seems, the difference between PlanetLab and other large distributed systems is only the size of the system at which such effects are observed, not the fundamental nature of the effects themselves.

Since it seems the problem of slow nodes may be endemic to large distributed systems, then, we believe the most appropriate response to it is to modify our system to be resilient to such behavior. We demonstrate our success with such techniques in Chapter 7.

1.4 Assumptions

Before continuing, we explicitly state two assumptions that limit the scope of this work.

First, we do not discuss the design or implementation of strongly consistent distributed systems. Instead, Bamboo and OpenDHT offer only eventually consistent semantics, in the style of Bayou [PST+97]. This approach is in line with the majority of the literature on DHTs, although there are notable exceptions [MGM05, LMR02, RL03]. We view the development of DHTs with stronger semantics as important and valuable work, but note that there are numerous applications for which eventual consistency is sufficient; the list in Table 5.4 provides several examples. As such, we leave stronger consistency to future work.

Second, a major benefit of DHTs is that they are applicable in situations in which there is no natural point of centralization. In the case where such a point exists, one can of course offer the same or similar semantics provided by a DHT with a centralized system, either by running a DHT on a cluster or by the use of some other architecture. As clusters grow to the scale of tens of thousands of nodes, the use of DHT-like techniques within them may become an interesting area of research. In this work, however, our goal is to simplify the construction of scalable systems using existing, distributed resources, and we consequently leave the case of centralized systems to future work.

1.5 Summary

Distributed hash tables are a promising building block on which more sophisticated distributed applications can be built. In this thesis we explore several important problems in their design, implementation, and deployment. We start in Chapter 2 by providing important background information related to DHTs. Next, we cover two fundamental challenges—the lookup and storage problems—in Chapters 3 and 4, respectively. The solutions to these two problems are implemented in Bamboo, the DHT we designed and built. We then explore how a single deployment of Bamboo can be shared as a public service among multiple applications, an idea we call OpenDHT, in Chapter 5. Running the OpenDHT service over time has also exposed us to several interesting practicalities that arise in running a large DHT deployment on the Internet (rather than in simulation, for example), and we cover two of these in Chapters 6 and 7. Finally, we conclude in Chapter 8.

Chapter 2

Background

This chapter provides an introduction to DHTs. To make the discussion concrete, we begin with a high-level description of the Bamboo DHT as an example. We then use this description as a starting point to review the advantages and limitations of DHTs, and we conclude the chapter with a survey of related work in the area.

2.1 The Bamboo DHT

The key space used by Bamboo is $\mathbb{Z}_{2^{160}}$, the integers modulo 2^{160}. Bamboo assigns each node a unique identifier $n \in \mathbb{Z}_{2^{160}}$ uniformly at random.[1] For convenience, we will name nodes by their identifiers. Furthermore, let $\mathrm{pred}(k)$ be the node whose identifier most immediately precedes k in $\mathbb{Z}_{2^{160}}$, and let $\mathrm{succ}(k)$ be the node whose identifier most immediately succeeds k. Bamboo partitions $\mathbb{Z}_{2^{160}}$ between the nodes in the DHT by mapping each key k onto the node n that minimizes $|k-n| \bmod 2^{160}$, or onto $\mathrm{succ}(k)$ if there are two such nodes. The node onto which k is mapped is called the *root* for k.

To maintain this mapping, each Bamboo node n keeps track of both $\mathrm{pred}(n)$ and $\mathrm{succ}(n)$; we call these two nodes its *predecessor* and *successor*. By knowing its predecessor and successor a node can compute exactly which keys it is responsible for under the mapping. Since the set of predecessor and successor links form a circular doubly-linked list between the nodes in the overlay, we often refer to the network as a *ring*.

The process of computing onto which node a key is mapped is called *lookup*. To perform

[1] In practice, we assign a node with IP address a listening on port p the identifier $H(a \cdot p)$, where H is the SHA-1 hash function and $\cdot$ represents concatenation.

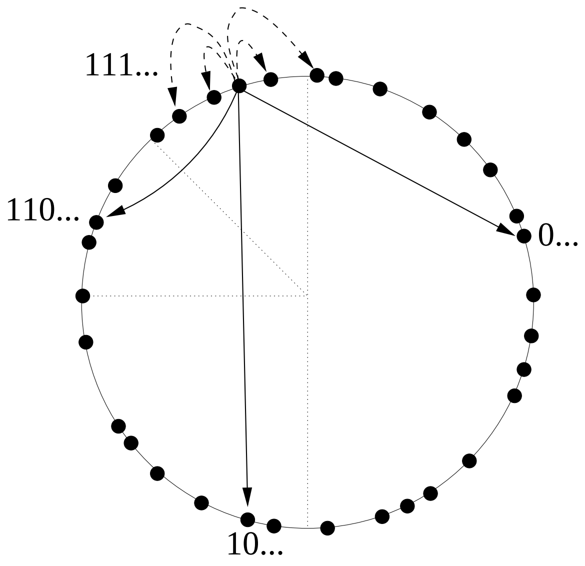


Figure 2.1: *A Bamboo node's neighbors.* A node's neighbors are divided into its *leaf set*, shown as dashed arrows, and its *routing table*, shown as solid arrows.

a lookup on key x, a node can simply route a message around the ring until it reaches the root for x, but this technique is neither efficient nor robust. It is inefficient because a lookup may be forwarded all the way around the ring before reaching the root. It is fragile since the loss of even a single node in the DHT will break the ring, rendering some lookups impossible. Bamboo fixes these two shortcomings through the use of two separate sets of neighbors maintained by each node. Both sets are illustrated in Figure 2.1.

The first set of these neighbors increases the robustness of the mapping by adding redundancy to the ring. Let $\text{pred}_i(k)$ be the result of applying pred to k i times (for $i > 0$), and let $\text{succ}_i(k)$ be defined similarly. The *leaf set* of a node n is the set of nodes $\text{pred}_i(n)$ and $\text{succ}_i(n)$ for $i \in [1, \ell]$. We call ℓ the *radius* of the leaf set.

The second set of neighbors a node maintains allows it to perform lookups more efficiently. A node's *routing table* is a set of nodes whose identifiers share successively longer prefixes with its own identifier. Given some base B, and for every prefix p of n, node n has a neighbor with prefix $p \cdot d$ for each digit $d \in [0, B)$, where $\cdot$ represents concatenation (if such a node exists). As illustrated in Figure 2.1, for $B = 2$ this method of choosing routing table entries corresponds to each node knowing some node on the opposite half of the ring, on the opposite quarter of its own half, on the opposite eighth of its own quarter, etc.

A Bamboo node performs a lookup on key k by matching as long a prefix of k as it can

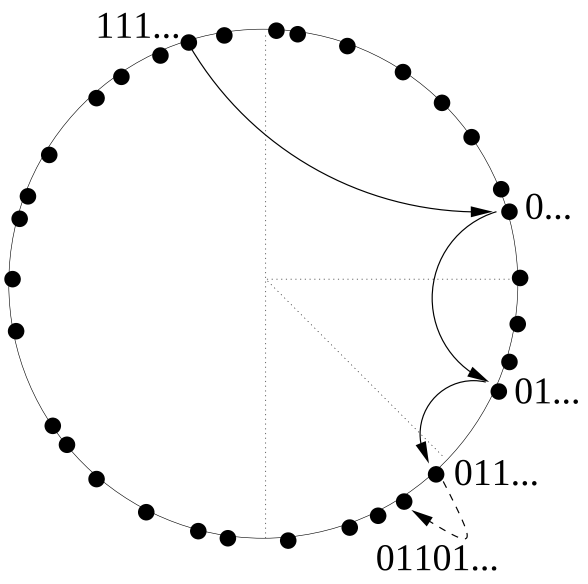


Figure 2.2: *A lookup in Bamboo.* To find the node closest to identifier 01101, the node whose identifier starts with 111 sends a lookup message to its neighbor whose first digit is 0. This node then forwards the query to its neighbor whose first two digits are 01, and from there the node is forwarded to the neighbor whose first three digits are 011. A final hop through a leaf set neighbor locates the closest node.

by following routing table links, then routes the additional distance to *k*'s root by following leaf set links. This process is illustrated in Figure 2.2.

To join an existing DHT, a new node *n* uses any existing node to find the root for *n*, from which it can retrieve its leaf set. Moreover, *n* and the root for *n* likely share a long prefix, in which case they will share many routing table entries. All other routing table entries can be filled by doing lookups on keys with the appropriate prefixes.

A simple implementation of the put function of a traditional hash table stores a key-value pair with key *k* the nodes $\text{pred}_i(k)$ and $\text{succ}_i(k)$ for $i \in [1, \ell']$, where $\ell' \leq \ell$, the leaf set radius. These nodes are easily discovered by doing a lookup on *k* to find its root, and then asking the root for its leaf set. Likewise, a simple implementation of get for a key *k* does a lookup on *k* to find the root and asks it for all the values it has stored under *k*.

2.2 Advantages of DHTs

With this simple description of Bamboo, we now enumerate the advantages of DHTs. First, we note that the system is completely decentralized. Any node is capable of performing a lookup, put, or get on any key, and to join the DHT a new node need only know of one other existing node. The importance of this advantage is illustrated by the fact that many applications in which we wish to use a DHT, such as Coral and FeedTree, are by their nature completely decentralized, allowing for no central point of organization.

Second, as long as $\ell = O(\log N)$, each node maintains only a logarithmic number of neighbors in the network. This feature of DHTs is important because in order to detect the failures of its neighbors, a DHT node must periodically probe them for liveness; the bandwidth usage of the DHT thus scales linearly in the number of neighbors per node.

Third, the cost of lookups, puts, and gets scales logarithmically in the size of the network. To see that this is the case, note that we expect to need $\log_B N$ digits to uniquely specify any given node, and that at any given point, there is a $\frac{B-1}{B}$ chance that the current node is a suitable next hop (i.e., it already has the required next digit). Since the cost of operations grows slowly in the size of the network, we can increase the capacity of the network by merely adding nodes.

Furthermore, looking back at Figure 2.1, we can see that for one neighbor, a node can choose from roughly half the nodes in the network. For another neighbor, it chooses between a quarter of the nodes, and so on. Proximity neighbor selection (PNS) is the technique by which a node chooses within each group to minimize the network latency between it and its neighbors. Dabek et al. [DLS$^+$04] have shown that using PNS the average latency of lookups can be made constant in the size of the network.

Fourth, the network is very robust. For $\ell > 1$, the ring can survive the failure of any arbitrary $\ell - 1$ nodes without a disconnection in the ring (i.e., the situation where some node either has no live predecessors or successors). Moreover, as shown by Stoica et al. [SMK$^+$01], if $\ell = O(\log N)$ and each node in the network fails is probability 1/2, with high probability no disconnection will occur. Likewise, any arbitrary $2\ell' - 1$ nodes can fail without data loss, and if all nodes fail with probability 1/2, no data will be lost with high probability so long as $2\ell' = O(\log N)$.

Finally, we note that the system is almost completely self-organizing and self-maintaining. When started, each node must somehow discover some other node through which to join the network, but no other configuration information is necessary.

2.3 Limitations of DHTs

As described by Blake and Rodrigues [BR03], there are three factors that limit the performance of a DHT: the total amount of data stored, the bandwidth available to each node, and the turnover in the system's membership. Note that the partitioning of the key space in a DHT is entirely a function of the nodes that comprise the system at any given time. As such, whenever a new node joins the system or an existing node fails, the partitioning changes, and data must be moved so that get requests continue to be routed to the nodes storing the desired data.

Let us consider the effects of new nodes joining and existing nodes leaving separately. As discussed above, DHTs store data redundantly for fault tolerance. Following Blake and Rodrigues, let us denote the factor of redundancy as r, the unique data stored in bytes as D, the total data stored in bytes as $S = rD$, the size of the system as N, and the average amount of time each node is part of the system in seconds as T.

When an existing node leaves the system, it takes the data it has stored with it. This redundancy must be restored by copying some data onto the remaining hosts. Overall, the amount of bandwidth used per node to handle such failures is S/NT bytes per second (see [BR03] for a derivation). We note that this cost is not unique to DHTs, but applies to *any* replicated storage system, and it implies that any such system must either have a relatively stable membership, a great deal of available bandwidth, or store only a small amount of data.

The cost of nodes joining, on the other hand, is unique to DHTs. Because of the strict partitioning that maps data items to nodes, DHTs also move data when a new node joins the system. Overall, the amount of bandwidth used per node to handle such failures is also S/NT bytes per second. This cost cannot be removed without changing the nature of the DHT, but as it is no larger than the cost of nodes leaving, it does not fundamentally alter the range of environments where DHTs are applicable.

So far we have considered only the cases where nodes join the system for the first time or leave the system forever. From this analysis, it is clear that to store a large amount of data in the DHT, nodes must remain part of the system for a long time or the bandwidth costs dominate. Under many circumstances this seems like a reasonable requirement; in the Coral system, for example, each participating web site will presumably remain part of the system for months or longer. It is unreasonable, however, to expect each of these sites to remain continuously available for the entirety of that time. Machines may crash, for example, or they may be rebooted after the installation of security patches.

To their peers, however, such temporary failures are indistinguishable from permanent ones, and a naive DHT implementation may trigger recovery mechanisms to account for them. If a temporary failure is indeed short lived, the bandwidth used during this unnecessary recovery is wasted. In Section 4.6.3 we discuss mechanisms that can be used to largely eliminate the cost of these temporary failures.

In summary, DHTs are suitable for two important domains. First, a DHT is appropriate for providing a small amount of data with a high degree of availability across a set of peers with dynamic membership. Examples of such applications include all those where a DHT is primarily used as an index, such as Coral, or where it is used primarily for communication, as in FeedTree. Second, a DHT can also be used to provide highly available access to a large data repository with low maintenance cost, but only if the membership of such a system is relatively stable. OpenDHT, OverCite [SCL+05], and UsenetDHT [SDR04] are all good examples of this latter use of a DHT.

2.4 Related work

We now provide an overview of related work in this area in order to give context to this thesis. Here we focus on the larger issues that distinguish our work from others'. We will cover more detailed differences where appropriate throughout the rest of this work.

2.4.1 DHT Geometries

The pattern of neighbor links in the overlay network of a DHT is commonly called its *geometry* [GGG+03]. In particular, the term geometry is used to speak specifically about this graph itself, rather than the graph maintenance or routing algorithms used by the DHT. Because DHTs with different geometries can use the same routing algorithm (e.g., greedy progress in the key space), geometry is a useful first metric with which to distinguish one DHT from another.

The Original DHT Geometries The first group of DHT geometries that were proposed all provided the same rough cost-performance tradeoff. Chord [SMK+01], Pastry [RD01], Tapestry [ZHS+04], and Kademlia [MM02] all use a graph where each node has $O(\log N)$ neighbors and a lookup operation takes $O(\log N)$ hops. CAN [RFH+01] uses a graph where each node has d neighbors and a lookup takes $O(dN^{1/d})$ hops; for $d = \log N$, a CAN node has $O(\log N)$ neighbors and takes $O(\log N)$ hops to perform a lookup. Bamboo provides the same tradeoff as these DHTs, as it uses the Pastry geometry. Of these original DHTs, the Chord, Pastry, and Kademlia geometries have found wide use in widely deployed systems [SAZ+02, SDR04, M+03, RGK+05, edo, bit].

Constant-State Geometries The next group of DHT geometries fell on different sides of this tradeoff. First, by using a geometry based on a de Bruijin graph, Kaashoek and Karger presented a DHT called Koorde that maintains a constant number of neighbors per node yet still performs lookups in a $O(\log N)$ hops [KK03]. Nonetheless, they showed that for fault-tolerance, it is still desirable that each node have $O(\log N)$ neighbors. In this case, Koorde performs lookups in $O(\log N/\log\log N)$ hops.

While Koorde is thus capable of performing lookups in less hops than the original group of DHTs, it has very little choice in neighbors, preventing it from selecting its neighbors for proximity. In contrast, the original DHTs (including Bamboo) use PNS to perform lookups in *time* constant in the size of the network (though still using a logarithmic number of hops) [DLS$^+$04]. While they are thus interesting from a theoretical point of view, we do not expect to see many DHTs based on de Bruijin graphs used in practice.

Constant-Hop-Count Geometries The other direction that DHT geometries have moved is towards using a larger number of neighbors to perform lookups in a number of hops constant in the size of the system. DHTs in this group include Kelips [GBL$^+$03] and the one-hop design of Gupta, Liskov, and Rodrigues [GLR04] (which we will subsequently call "OneHop" for brevity).

Kelips divides the membership of a DHT into $k = O(\sqrt{n})$ *affinity groups*. Each node maintains state about all the nodes in its own affinity group, as well as state about a few nodes in each other affinity group. Lookups can thus be performed in two hops—one to the correct affinity group and one within that group. To manage this larger number of neighbors efficiently, Kelips uses epidemic propagation both within and between groups. When this state becomes stale, Kelips may take more than two hops to resolve a lookup, but simulations show that this case can be limited. The designers of Kelips argue that it should scale to around 100,000 nodes.

The OneHop algorithm goes even further than Kelips by maintaining state about *every* node in the network at every other node. The trick, of course, is how to propagate that information efficiently. Like Kelips, the OneHop algorithm uses epidemic propagation of membership changes. Compared to Kelips, however, its propagation graph is more structured. The OneHop system is divided into $k = O(\sqrt{n})$ slices, which are roughly equivalent to Kelips' affinity groups. Each slice has a *slice leader*, and communication between slices occurs only between leaders. Each slice is further subdivided into several units, each of which has a *unit leader*, and communication between slice leaders and the nodes in the slice occurs only through unit leaders. Simulations show that the OneHop design uses reasonable bandwidth, although the bandwidth requirements for slice leaders indicate that they should be chosen carefully. The OneHop authors also present a two-hop design using the same geometry that they believe will scale to a few million nodes.

Despite their apparent promise, we are not aware of any deployed systems that use constant-hop DHTs such as Kelips or the OneHop algorithm.

A Variable-State Geometry The DHT geometries above present a tradeoff between the bandwidth used in maintaining neighbor links versus the number of hops required to perform a lookup. In situations where bandwidth is scarce, the constant-state geometries are more attractive. In contrast, constant-hop geometries are preferred in situations where bandwidth is ample. A natural question thus arises as to whether it is possible to design a DHT geometry that adapts to the bandwidth available to deliver the fewest hops possible in any situation. Accordion [LSMK05] is a proposed design for a variable-state geometry of this form. While it has only been simulated to date, early results are promising.

2.4.2 Lookup Practicalities

The geometry of a DHT is only one component of its performance in practice; the routing algorithm used is often just as important. There are two practical issues that make routing algorithms vital: the computation of timeouts on routing messages, and the non-uniform cost of hops in the network.

Timeout calculation In a any large distributed system, the sheer number of nodes ensures that at any given time some nodes will be down, others will be in the process of crashing, and still others will just be slow (due to temporary load, faulty components, etc.). Furthermore, studies of existing peer-to-peer systems show even higher rates of failure than other, similarly sized distributed systems [SGG02, CLL02, SW02, BSV03, GDS$^+$03]. If failures are common, and detecting and routing around a failed node takes several seconds, the cost of failures can easily come to dominate the cost of a lookup.

In Chapter 3 we describe our work for handling failures along the lookup path in DHTs. Our primary observation is that through continuous, active probing of its neighbors in the graph, a DHT node can compute good values for the time it expects each neighbor to take to process a lookup message. Messages that are not acknowledged within this period can quickly be resent along an alternate path.

In practice, to adequately guard against false positives, message timeouts generally need to be some small multiple of average round-trip time, and even a single timeout can add significantly to query latency. One approach to limiting the effect of timeouts on end-to-end lookup latency is to

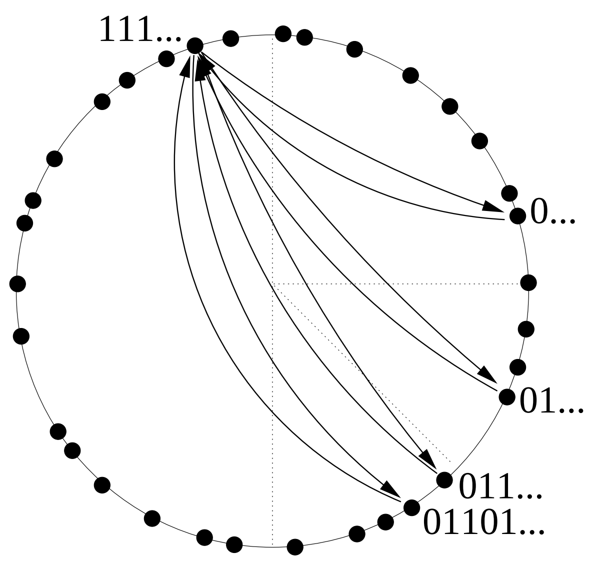


Figure 2.3: *Iterative lookup.* An iterative lookup involves the same nodes as a recursive one, but instead of forwarding the message, each intermediate node responds to the source with the address of the next hop.

parallelize the lookup; by routing along several paths simultaneously, a DHT can use extra resources to increase the number of timeouts that must occur in order to stall a lookup.

A simple way to parallelize lookups is to issue each lookup from multiple source nodes in the DHT. As discussed in Chapter 7, the effect of such parallelization is dramatic in our OpenDHT deployment on PlanetLab.

The lookup process we have described so far (and illustrated in Figure 2.2) is commonly called *recursive lookup*. An alternate lookup algorithm that is useful for parallelization is called *iterative lookup*. As illustrated in Figure 2.3, an iterative lookup contacts the same nodes in the DHT as a recursive lookup for the same key, but the lookup process is directed at all points by the source node; the lookup is not routed through the DHT. Since one node is in charge of the lookup at all times, it is easy to parallelize: the source node just keeps several RPCs active at any time. This approach to lookup was first proposed in Kademlia, and we show its effectiveness in Chapter 7.

Although parallel iterative lookup can limit the effect of timeouts on the lookup as a whole, it introduces a new problem: since the nodes contacted in during the lookup process are often not the immediate neighbors of the source node, it is not immediately clear how to compute timeout values for messages sent to them. Dabek et al. [DLS$^+$04] showed that network coordinates computed using the Vivaldi [CDK$^+$03b] algorithm are appropriate for this purpose. In Chapter 3 we compare the quality of timeouts computed using Vivaldi versus those computed by direct measurement.

Lookup Hop-Count vs. Latency Our earlier categorization of DHT geometries above focused primarily on the number of *hops* required to perform a lookup. From the point of a user, however, a more natural metric is the end-to-end *latency* of a lookup. In other words, it is not just the number of hops that should be considered, but also the latency of each hop.

By the metric of end-to-end latency, it is not immediately clear that greedy routing in the identifier space is optimal. DHash [DLS+04] uses a variant of Chord routing where each node picks its next hop by computing the expected latency to complete a lookup through each neighbor and chooses the neighbor that minimized this latency. This estimation is based on the latency to the neighbor itself, plus the average latency in the overlay times the expected number of hops that would remain after contacting that neighbor. The number of hops remaining is estimated based on the observed density of the nodes in the identifier space.

Gummadi et al. [GGG+03] explored a technique the called *proximity route selection* (PRS), whereby a lookup query was routed at each hop to the neighbor closest in network latency that made progress in the identifier space. The authors show that PRS outperforms greedy routing in a simulation of Chord using a realistic network latency distribution, but they did not account for the additional cost of processing at each node. As the PRS route is likely to involve more hops than the greedy one, this per-hop processing cost can be significant in real deployments.

In Chapter 7, we present the results of experiments that explore the value of PRS in the OpenDHT deployment on PlanetLab. We show that while PRS does indeed provide some benefit, it is not optimal. On PlanetLab, per-hop processing costs are non-negligible, so a hybrid of greedy and proximity-based routing works best. This hybrid routes greedily with respect to low-latency neighbors, but weights proximity more heavily for high-latency ones.

2.4.3 DHT Storage

As discussed in the introduction, while many DHT applications use the relatively primitive lookup interface, others want a higher-level interface to the DHT. One example of such an interface is the traditional put/get interface offered by hash tables. The implementation of put/get atop lookup is conceptually simple: to put, a node looks up the root of a key and sends it a put message; to get, it looks up the root and sends it a get message. As with lookup, however, the difficulty of implementing put/get in practice is in handling failures. The main additional functionality of put/get over lookup is thus the fault-tolerant storage of key-value pairs. As other DHT interfaces (e.g., DOLR [ZHS+04, DZD+03]) also require storage within the DHT, we refer to the problem in general as the storage problem. There are two main approaches in the literature to handling it.

The Soft-State Approach The first approach, which we will call the *soft-state* approach, places the responsibility for maintaining the availability of each key-value pair outside of the DHT. To keep its key-value pairs available, a client of a soft-state DHT must re-put them before all of the nodes onto which they were originally replicated fail. Moreover, it must also re-put them when enough new nodes join the DHT such that it no longer maps the key onto any of the nodes where replicas were originally stored.

Despite these complications, the soft-state approach is very easy to implement in the DHT, and it is therefore attractive in applications where the burden on clients is small. For example, in *i*3 [SAZ+02], a DHT is used to forward packets to clients outside the DHT. To enable this functionality, a client puts its IP address into the DHT under a particular key. Other clients pass messages to the DHT with this key, and the DHT forwards them to whatever IP address is currently stored under that key. Because each client only has one IP address, it can re-put that address frequently at low cost. Furthermore, the DHT can garbage collect old values by expiring any addresses not put in the last minute or so.

In other cases, the soft-state approach is less attractive. For example, a Coral node may have a very large web cache, and it would take considerable cost to re-put each of the entries in its cache index into the DHT every minute. Alternatively, it could keep track of which DHT nodes stored its values and monitor them, but this approach still scales at best linearly in the size of the DHT or the number of values, whichever is larger.

The Managed Approach A more efficient solution in such cases is to have the replicas for any one value monitor each other, an approach we call the *managed* approach. In this approach, the DHT is responsibly for storing data redundantly and for restoring that redundancy after failures. By the nature of the DHT's partitioning function, nodes whose identifiers are adjacent in the key space are replicas for many of the same values. This arrangement presents the possibility of a maintenance protocol whose total bandwidth usage is proportional to the number of nodes in the system, a marked improvement from common implementations of the soft-state approach, which use bandwidth proportional to the number of values stored.

The managed approach to DHT storage was first proposed by both the Cooperative File

System [DKK$^+$01] and PAST [DR01], two early DHT storage systems, but neither presented an efficient solution for implementing it.

In Chapter 4 we present an algorithm for storage management in which each node uses only a constant amount of bandwidth per unit time when all r replicas for a set of values are in sync, and it uses at worst a cost of $O(r \log n)$ to find a missing value on one of the nodes in such a set, where n is the total number of values replicated. Moreover, in many cases it discovers many such inconsistencies at once, amortizing this $O(r \log n)$ cost across the reconciliation of many replicas. This algorithm was developed concurrently with, but independently of, that in Cates' thesis [Cat03]; we discuss the differences in detail in Chapter 4.

An interesting variation on the managed approach is used in the Beehive system [RS04b], which replicates data very widely in order to reduce get latency. Using a distribution of the popularity of items, Beehive replicates each item such that the average number of hops performed per get in the system is below 1. This approach requires a great deal of replication; one example in [RS04b] suggests a replication factor of 37. The cost of this replication is born not only by the storage resources of the system, but also by the network resources, as each update to a replicated item must be propagated to each replica. Nevertheless, the resulting improvements in latency may be justified in a system storing a small number of items that change relatively rarely.

2.4.4 Sharing A DHT Between Applications

As discussed in the introduction, there are two senses in which a DHT can be shared: among applications and among clients. We discuss sharing between applications first.

Facilitating bootstrap As discussed in the introduction, by using the same DHT for all feeds, the FeedTree application need only bootstrap itself once; knowing a client for a specific feed is sufficient to find clients for any other feed. This bootstrap problem extends to DHTs in general: in order to join a particular DHT, a new node must know of at least one existing node in that DHT. To discover such a node, some form of lookup service is needed, yet scalable lookup is exactly the service provided by DHTs.

A common solution to the bootstrap problem uses the DNS system to locate existing nodes. In this solution, a small set of DNS servers are registered as the name servers for a domain. These same servers monitor the live nodes in the DHT, and they use this monitoring information to return only live nodes' IP addresses as address records in response to DNS queries. This technique is used in Coral, for example. Unfortunately, most kernel DNS clients will only access remote DNS servers on port 53, limiting the number of DNS servers that can run on a shared testbed such as PlanetLab. It is thus difficult for every DHT-based application to solve the bootstrap problem using DNS.

Another approach to solving the bootstrap problem is the idea of sharing a single DHT between all DHT applications. The earliest such example of which we are aware is the “One Ring” proposal of Castro et al. [CDKR02]. In this proposal, each node in a DHT is both a member of its application-specific DHT *and* a single, global DHT. This global DHT is used to provide a number of services, including a file store, multicast, and a bootstrap service for the application-specific DHTs. Essentially, the IP addresses of several bootstrap nodes are stored under a service-specific key for each application DHT, and new nodes lookup these bootstrap nodes to join a given ring.

The One Ring approach eliminates the need to use DNS for each application-specific DHT, but it does not balance the bootstrapping load well. Only the small set of nodes registered with the global DHT are available as bootstrap hosts, and the node (in the global ring) that stores this list for each application is saddled with all of the load for looking up this set of bootstrap nodes. While caching can be used to eliminate some of this load, it will also reduce the freshness of the information returned.

Two algorithms designed to address the limitations of the One Ring approach have been proposed: Karger and Ruhl proposed Diminished Chord [KR04], and we proposed Recursive Distributed Rendezvous (ReDiR) [KRRS04].

Diminished Chord is a variant of the Chord protocol that allows several Chord applications to share a single Chord ring. In this design, the lookup function is parameterized by application; a lookup for key k in application a finds the node in the ring running application a that most immediately succeeds k in the identifier space. As with the standard Chord algorithm, these lookups take only $O(\log N)$ hops in a global network of N nodes, but unlike the standard algorithm, the graph for Diminished Chord cannot be built using proximity neighbor selection. In practice, then, we expect lookups in Diminished Chord to be slower than those in standard Chord using PNS.

In contrast to Diminished Chord, the ReDiR algorithm is a client-side library that uses only the put and get functions of any DHT implementation to provide an application-specific lookup function. In the worst case, this lookup requires $O(\log N)$ gets to find the successor of a given key, but on average it requires only a constant number of gets to do so. Since these gets can also be optimized using PNS, we expect ReDiR to be at least as fast as Diminished Chord in practice. ReDiR is discussed in more detail in Chapter 5.

In comparison to the One Ring proposal, both Diminished Chord and ReDiR spread the bootstrapping load for each application across all of the nodes in the global DHT and all of the nodes in the application. As such, once a node can access the global ring, the bootstrapping process for any individual application is just as scalable as the remainder of the DHT.

The DHT as a service Techniques like those of the One Ring, Diminished Chord, or ReDiR make deploying a new DHT application easier by solving the bootstrap problem. Nonetheless, a new DHT application must still deploy some nodes; if there is not at least one node online at any time, there will be no one for new nodes to bootstrap through. While it seems somewhat ridiculous that there could exist an interesting application that both needed the scalability of a DHT and simultaneously could experience periods of without any active members, practical issues make the idea not as far fetched as it sounds. For example, what if all the active members of an application are behind NATs, and thus unable to contact each other?

To address this problem, we take an even more radical approach than those described above: by running a DHT as an Internet *service*, we make it possible to build DHT-based applications without deploying any DHT nodes at all. We discuss the resulting system, called OpenDHT, in detail in Chapter 5.

2.4.5 Sharing A DHT Between Clients

In sharing a DHT among clients, the primary difficulty is that of resource allocation. A faulty or malicious client, for example, may perform enough puts to fill the DHT's storage resources or enough gets to overwhelm the bandwidth available to some nodes in the DHT. If this situation is not prevented, other clients will experience reduced service. While there are known techniques for allocating computation and network resources (e.g. [DKS89, DC99, NL97]), the allocation of storage is less well understood.

One early approach to storage allocation in a shared system was introduced in the Palimpsest system [RH03]. Palimpsest uses a novel revolving-door technique in which, when the disk is full, new stores push out the old. To keep their data in the system, clients re-put frequently enough so that it is never flushed; the required re-put rate depends on the total offered load on that storage node. Palimpsest uses per-put charging, which in this model becomes an elegantly simple form of congestion pricing to provide fairness between users (those willing to pay more get more).[2]

[2]Referring back to Section 2.4.3, we note that Palimpsest presents a novel reason for using soft-state storage management in a DHT.

While we agree with the basic premise that public storage facilities should not provide unboundedly persistent storage, we are reluctant to require clients to monitor the current offered load in order to know how often to re-put their data. This adaptive monitoring presents the same complications as the soft-state model of DHT storage discussed above. Moreover, Palimpsest relies on charging to enforce some degree of fairness; since OpenDHT is currently deployed on PlanetLab, an environment where such charging is both impractical and impolitic, we wanted a way to achieve fairness without an explicit economic incentive. Our solution to this challenge, called Fair Space-Time (FST), is discussed in Chapter 5.

Another system that is exploring the notion of shared storage in the wide area is the Internet Backplane Protocol (IBP) [BMP03]. While this system is not based on a DHT, our FST algorithm is applicable to it as well. Furthermore, we hope that the existence of the IBP system will encourage the development of alternate storage allocation algorithms that we can incorporate into OpenDHT.

2.4.6 Load Balancing

To close this section, we discuss one additional area of work in the DHT space. It is well known that when the identifiers of nodes in a DHT are chosen uniformly at random, the load imbalance between nodes can be $O(\log N)$ [SMK+01]. In other words, some nodes in the DHT will be responsible for a logarithmic factor more of the key space than others. Even if clients balance their key choices around the ring, then, some nodes in the DHT will see much higher storage and routing load than others. In the case where clients of the DHT do not balance their key choices, the problem is only exacerbated.

Load balancing in a DHT is the process of trying to modify the DHT so that each node is assigned either an equal share of the key space, the data stored, or the routing load. There are a number of algorithms in this space, and they rely on a variety of techniques. Ruhl and Karger's algorithm balances load by changing nodes' identifiers to more equally share the key space or storage load [RK04]. PAST and the algorithm of Suri et al. [STZ04] balance load by moving large objects to underutilized nodes. Godfrey et al. [GLS+04] balance load by assigning several *virtual nodes* to each physical node in the DHT, as first proposed in Chord [SMK+01].

Since the amount of data stored by a node influences the bandwidth it uses on data maintenance, and since bandwidth is the limiting factor in the growth many DHTs, storage is often the most important resource to balance. Unfortunately, load balancing itself requires bandwidth (as data is moved), and so it cannot fully eliminate the problem. As we discuss in Chapter 5, we are thus in favor of encouraging application behavior that balances the storage used at the time it is allocated. Our FST algorithm, for example, rewards clients that can choose keys so as to place their values on underutilized nodes. Clients thus have a selfish incentive to target lightly loaded nodes, potentially ameliorating the need for later load balancing by the DHT itself.

2.5 Summary

In this chapter we have presented an introduction to the field of DHT research, discussed which portions of that field are covered in this thesis, and distinguished our work from the related work of others. In subsequent chapters we will cover each of our contributions in detail. To review, these include DHT lookup and storage in Chapters 3 and 4, the DHT as a service in Chapter 5, and several practical issues we have discovered in building and deploying a DHT in Chapters 6 and 7.

Chapter 3

Lookup

As discussed in the introduction, the most basic functionality of a DHT is *lookup*—the mapping of keys onto nodes in the DHT. This functionality is the lowest-level interface in the proposed Common API for DHTs [DZD+03], and it is the basic functionality on which the put/get [DKK+01,Cat03,RGK+05], multicast [ZZJ+01,RKCD01,RHKS01], and DOLR [ZHS+04] interfaces are built. It is thus of utmost importance that the lookup functionality be robust in all of the scenarios in which DHTs are to be deployed.

Early work on DHTs focused on large-scale failures. For example, several papers have shown that a DHT with a logarithmic number of neighbors per node can survive even when every node in the system fails with probability 1/2 [SMK+01, KK03].

At the same time, research into existing (but not necessarily DHT-based) peer-to-peer systems has shown that these networks are plagued not by large-scale simultaneous failures, but instead suffer from a steady and continuous process of nodes joining and leaving the system, a process we call churn. One measurement of churn, the median time between when a node joins the network and when it next departs, has been observed to be as long as an hour to as short as a few minutes in deployed systems [BSV03, CLL02, GDS+03, SGG02].

In this chapter we explore the performance of DHT lookup in such dynamic environments. DHTs may be better able to locate rare files than existing unstructured peer-to-peer networks [LHSH04]. Moreover, it is not hard to imagine that other proposed uses for DHTs will show similar churn rates to file-sharing networks—application-level multicast of a low-budget radio stream, for example. In spite of this promise, we show that high churn causes a variety of negative effects on two mature DHT implementations we tested. Both systems exhibit dramatic latency growth when subjected to increasing churn, and in one implementation the network eventually partitions, causing subsequent lookups to return inconsistent results. The remainder of this chapter is dedicated to determining whether a DHT can be built such that it continues to perform well as churn rates increase.

In fact, we demonstrate that DHTs can perform lookups at high churn rates, and we identify and explore several factors that affect the behavior of DHTs under churn. The three most important factors we identify are:

- reactive versus periodic recovery from failures
- calculation of message timeouts during lookups
- choice of nearby over distant neighbors

By *reactive recovery*, we mean the strategy whereby a DHT node tries to find a replacement neighbor immediately upon noticing that an existing one has failed. We show that under bandwidth-limited conditions, reactive recovery can lead to a positive feedback cycle that overloads the network, causing lookups to have high latency or to return inconsistent results. In contrast, a DHT node may recover from neighbor failure at a fixed, periodic rate. We show that this strategy improves performance under churn by allowing the system to avoid positive feedback cycles.

The manner in which a DHT chooses timeout values during lookups can also greatly affect its performance under churn. If a node performing a lookup sends a message to a node that has left the network, it must eventually timeout the request and try another neighbor. We demonstrate that such timeouts are a significant component of lookup latency under churn, and we explore several methods of computing good timeout values, including virtual coordinate schemes as used in DHash [DLS$^+$04].

Finally, we consider *proximity neighbor selection* (PNS), where a DHT node with a choice of neighbors tries to select those that are nearest to itself in network latency. We compare several algorithms for discovering nearby neighbors—including algorithms similar to those used in the Chord, Pastry, and Tapestry DHTs—to show the tradeoffs they offer between latency reduction and added bandwidth.

In performing this study, we build the Bamboo DHT [RGRK03], which was initially based on Pastry, as described in Chapter 2. Furthermore, we augmented Bamboo such that it can be configured to use any of the design choices described above. As such, we can examine each design decision independently of the others. Moreover, we examine the performance of each configuration by running it on a large cluster with an emulated wide-area network. This methodology is particularly important with regard to the choice of reactive versus periodic recovery as described above. Existing studies of churn in DHTs (e.g., [CCR03a, CJK$^+$03, LSG$^+$04, MCR03]) have used simulations that—unlike our emulated network—did not model the effects of network queuing, cross traffic, or message loss. In our experience, these effects are primary factors contributing to DHTs' inability to perform lookups quickly and correctly under churn. Moreover, our measurements are conducted on an isolated network, where the only sources of queuing, cross traffic, and loss are the DHTs themselves; in the presence of heavy background traffic, we expect that such network realities will exacerbate the ability of DHTs to handle even lower levels of churn.

Of course, this study has limitations. Building and testing a complete DHT implementation on an emulated network is a major effort. Consequently, we have limited ourselves to studying a single DHT on a single network topology using a relatively simple churn model. Furthermore, we have not yet studied the effects of some implementation decisions that might affect the performance of lookups under churn, including the use of alternate routing table neighbors as in Kademlia and Tapestry, or the use of iterative versus recursive routing.[1] Nevertheless, the Bamboo DHT described here has since been used as the base system for OpenDHT (described in later chapters), which has been running successfully on PlanetLab for over a year now.

The rest of this chapter is structured as follows: in Section 3.1, we review existing studies of churn in deployed file-sharing networks, describe the way we model such churn in our emulated network, and quantify the performance of mature DHT implementations under such churn. In Section 3.3, we study each of the factors listed above in isolation, and describe how Bamboo uses these techniques. In Section 3.4, we survey related work, and in Section 3.5 we discuss important future work. We conclude in Section 3.6.

3.1 The Problem of Churn

There have been very few large-scale, DHT-based application deployments to date, and so it is hard to derive good requirements on churn-resilience. However, P2P file-sharing networks provide a useful starting point. These systems provide a simple indexing service for locating files on those peer nodes currently connected to the network, a function that can be naturally mapped onto a DHT-based mechanism. For example, the Overnet file-sharing system uses the Kademlia DHT to

[1]We have studied the differences between iterative and recursive routing in our PlanetLab deployment (see Chapter 7), but not under the heavy churn rates used in this chapter.

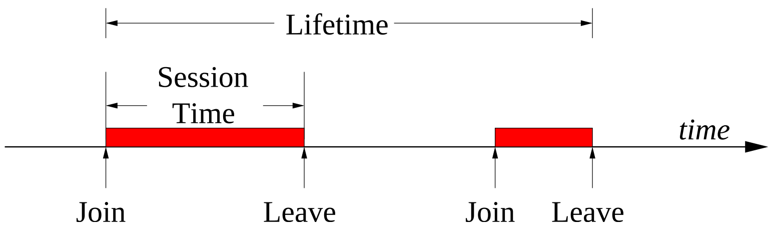


Figure 3.1: *Metrics of churn.* With respect to lookup, the *session times* of DHT nodes are more relevant than their *lifetimes*.

store such an index. While some DHT applications (such as file storage as in CFS [DKK+01]) might require greater client availability, others may show similar churn rates to file-sharing networks (such as end-system multicast or a rendezvous service for instant messaging). As such, we believe that DHTs should at least handle the churn rates observed in today's file-sharing networks. To that end, in this section we survey existing studies of churn in deployed file-sharing networks, describe the way we model such churn in our emulated network, and quantify the performance of mature DHT implementations under such churn.

Studies of existing file-sharing systems mainly use two metrics of churn (see Figure 3.1). A node's *session time* is the elapsed time between when it joins the network and when it subsequently leaves. In contrast, a node's *lifetime* is the time between when it enters the network for the first time and when it leaves the network permanently. The sum of a node's session times divided by its lifetime is often called its *availability*. One representative study [BSV03] observed median session times on the order of tens of minutes, median lifetimes on the order of days, and median availability of around 30%.

With respect to the lookup functionality of a DHT, we argue that session time is the most important metric. Even temporary loss of a routing neighbor weakens the correctness and performance guarantees of a DHT, and unavailable neighbors reduce a node's effective connectivity, forcing it to choose suboptimal routes and increasing the destructive potential of future failures. Since nodes are often unavailable for long periods, remembering neighbors that have failed is of little value in performing lookups.[2]

First Author	Systems Observed	Session Time
Saroiu [SGG02]	Gnutella, Napster	50% $\leq$ 60 min.
Chu [CLL02]	Gnutella, Napster	31% $\leq$ 10 min.
Sen [SW02]	FastTrack	50% $\leq$ 1 min.
Bhagwan [BSV03]	Overnet	50% $\leq$ 60 min.
Gummadi [GDS$^+$03]	Kazaa	50% $\leq$ 2.4 min.

Table 3.1: *Observed session times in various peer-to-peer systems.* The median session time ranges from an hour to a minute.

3.1.1 Empirical studies

Here we briefly survey five studies of existing peer-to-peer systems. The studies' findings are summarized in Table 3.1.

Saroiu, Gummadi, and Gribble [SGG02] presented the earliest study we have found of session times in peer-to-peer systems. Using active probing, they found the median session time in both Napster and Gnutella to be around 60 minutes. Another active study of Napster and Gnutella by Chu, Labonte, and Levine [CLL02] found that 31% of observed sessions were shorter than 10 minutes, and less than 5% were longer than 60 minutes. On the other hand, they observed a small fraction of sessions (less than 0.01%) lasting thousands of minutes at a time.

Sen and Wang [SW02] used passive monitoring to observe FastTrack traffic using routers in an ISP backbone. To compute session length, they included all traffic less than 30 minutes apart from the same IP address, and found that 60% of nodes had a total session time of under 10 minutes daily.

Bhagwan, Savage, and Voelker [BSV03] performed an active study of the Overnet system. The choice is significant since nodes in Overnet are uniquely identified by names that persist across sessions. As such, these names are more suitable for many metrics than IP addresses which vary over time due to DHCP, firewalls, etc. While this distinction is important for measuring node lifetimes, changing IP address involves leaving and rejoining a network, so we believe the previous studies' session time results are still valid.

Since the Overnet study did not include session times, we re-analyzed their data to extract them. This data contains the results of active probes for 2,400 distinct Overnet hosts every 20 minutes over a week. Marking the start of a session as the transition from a host being unreachable to being reachable, or as the change from one IP address to another, we found a median session time

[2] While remembering neighbors is useful for applications like storage [BR03], our point here is that it is of little value for *lookup* operations.

of 60 minutes, plus or minus the 20 minute probe period.

A study of Kazaa by Gummadi et al. [GDS+03] used passive measurement techniques to estimate session times as the length of continuous periods during which a node was actively retrieving files. They found a median session length of only 2.4 minutes, and a 90th percentile session length of 28.25 minutes.

Looking at the summary of observed session times in Table 3.1, we conclude that to replace existing systems, a DHT must be robust to session times from as long as an hour to as short as a minute on median. At first sight, the lower end of this range seems surprising, and may be due to methodological problems with the studies in question or malfunctioning of the system under observation. However, it is easy to image a user joining the network, downloading a single file (or failing to find it), and leaving, making session times of a few minutes at least plausible.

We further note that neither the studies we have cited nor our analysis take into account the possibility that sessions are cut short due to network failures, or that a robust DHT would experience longer session times due to its own resilience. Nevertheless, we feel that our derived requirements are a useful starting point for DHT designers.

3.1.2 Experimental Methodology

Our platform for measuring DHT performance under churn is a cluster of 40 IBM xSeries PCs, each with Dual 1GHz Pentium III processors and 1.5GB RAM, connected by Gigabit Ethernet, and running either Debian GNU/Linux or FreeBSD. We use ModelNet [VYW+02] to impose wide-area delay and bandwidth restrictions, and the Inet topology generator [ine] to create a 10,000-node wide-area AS-level network with 500 client nodes connected to 250 distinct stubs by 1 Mbps links. To increase the scale of the experiments without overburdening the capacity of ModelNet (by running more client nodes), each client node runs two DHT instances, for a total of 1,000 DHT nodes.

Our control software uses a set of wrappers that communicate locally with each DHT instance to send requests and record responses. Running 1,000 DHT instances on this cluster (12.5 nodes/CPU) produces CPU loads below one, except during the highest churn rates. Ideally, we would measure larger networks, but 1,000-node systems already demonstrate problems that will surely affect larger ones.

In an experiment, we first bring up a network of 1,000 nodes, one every 1.5 seconds, each with a randomly assigned gateway node to distribute the load of bootstrapping newcomers. We then churn nodes until the system performance levels out; this phase normally lasts 20-30 minutes but can take an hour or more. Node deaths are timed by a Poisson process and are therefore uncorrelated and bursty. A new node is started each time one is killed, maintaining the total network size at 1,000. This model of churn is similar to that described by Liben-Nowell et al. [LNBK02]. In a Poisson process, an event rate λ corresponds to a median inter-event period of $\ln 2/\lambda$. For each event we select a node to die uniformly at random, so each node's session time is expected to span N events, where N is the network size. Therefore a churn rate of λ corresponds to a median node session time of

$$t_{\text{med}} = N \ln 2/\lambda.$$

For example, a 1,000-node network churning with median session times of one hour will see one node arrive (and one leave) every 5.2 seconds. In our experiments, we used churn rates ranging from 8 joins/departures per second to 4 per minute, equal to median session times from 1.4 minutes to 3 hours.

During an experiment, each live node continually performs lookups for identifiers chosen uniformly at random, timed by a Poisson process with rate 0.1/second, for an aggregate system load of 100 lookups/second. Each lookup is simultaneously performed by ten nodes, and we report both whether it completes and whether it is consistent with the others for the same key. If there is a majority among the ten results for a given key, all nodes in the majority are said to see a consistent result, and all others are considered inconsistent. If there is no majority, all nodes are said to see inconsistent results. This metric of consistency is more strict than that required by some DHT applications. However, both MIT's Chord implementation and our Bamboo implementation show at least 99.9% consistency under 47-minute median session times [RGRK03], so it does not seem unreasonable.

There are two ways in which lookups fail in our tests. First, we do not perform end-to-end retries, so a lookup may fail to complete if a node in the middle of the lookup path leaves the network before forwarding the lookup request to the next node. We observed this behavior primarily in FreePastry as described below. Second, a lookup may return inconsistent results. Such failures occur either because a node is not aware of the correct node to forward the lookup to, or because it erroneously believes the correct node has left the network (because of congestion or poorly chosen timeouts). All DHT implementations we have tested show some inconsistencies under churn, but carefully chosen timeouts and judicious bandwidth usage can minimize them.

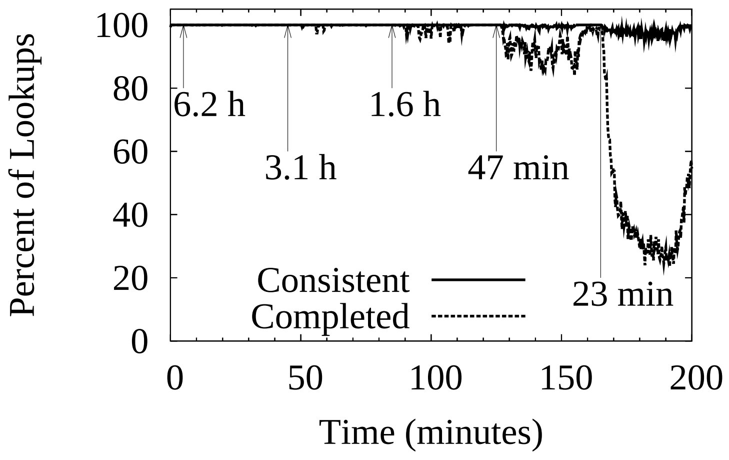


Figure 3.2: *FreePastry under churn.* The percentage of successful lookups in a 1,000-node FreePastry network under churn. Session times for each 30-minute churn period are indicated by arrows, and each churn period is separated from the next by 10 minutes of no churn. The churn rate doubles with each successive period.

3.1.3 Existing DHTs

In this section we report the results of testing two mature DHT implementations under churn. Our intent here is not to place a definitive bound on the performance of either implementation. Rather, it is to motivate our work by demonstrating that handling churn in DHTs is a non-trivial problem. While we have discussed these experiments extensively with the authors of both systems, it is still possible that alternative configurations could have improved their performance. Moreover, both systems have seen subsequent development, and newer versions may show improved resilience under churn.

FreePastry We tested FreePastry 1.3, the Rice University implementation of Pastry [fre]. Figure 3.2 shows one effect of churn on a network of 1,000 FreePastry nodes, which we ran using the default 24-node leaf sets and logarithm base of 16. We do not enforce proximity between a new node and its gateway, as suggested for best FreePastry performance; this decision only effects the proximity of a node's neighbors, not the efficiency of its routing.

It is clear from Figure 3.2 that while successful lookups are mostly consistent, FreePastry fails to complete a majority of lookup requests under heavy churn. A likely explanation for this failure is that nodes wait so long on lookup requests to time out that they frequently leave the network with several requests still in their queues. This behavior is probably exacerbated by FreePastry's

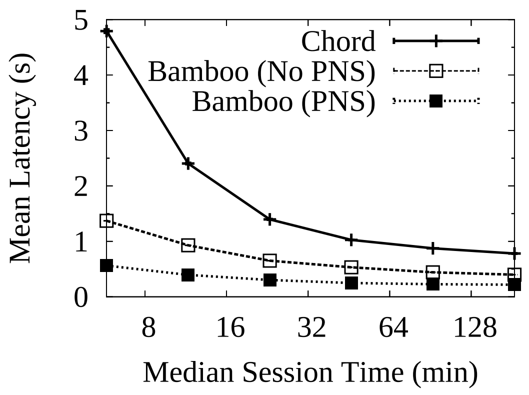


Figure 3.3: *Chord under churn.* Shown is the mean latency of lookups in a 1,000-node MIT Chord network under increasing levels of churn. Churn increases to the left.

use of Java RMI over TCP as its message transport and the way that FreePastry nodes handle the loss of their neighbors. We present evidence to support these ideas in Section 3.3.1.

Also note that FreePastry generally recovers well between churn periods, once again correctly completing all lookups. The difficulty with real systems is that there is no such quiet period; the network is in a continual state of churn.

MIT Chord We tested MIT's Chord implementation [mit] using a CVS snapshot from 8/4/2003, with the default 10-node successor lists and with the location cache disabled (using the `-F` option), since the cache causes poor performance under churn.

In contrast to FreePastry, almost all lookups in a Chord network complete and return consistent results. Instead, Chord's shortcoming under churn is in lookup latency, as shown in Figure 3.3, which shows the result of running Chord under the same workload as shown in Figure 3.2, but where we have averaged the lookup latency over each churn period. Shown for comparison are two lines representing Bamboo's performance in the same test, with and without proximity neighbor selection (PNS). Under all churn rates, Bamboo's bandwidth usage is slightly under 750 bytes per second per node, while Chord's is slightly under 2,400.

We discuss in detail the differences that enable Bamboo to outperform Chord in Sections 3.3.2 and 3.3.3, but some of the difference in latency between Bamboo and Chord is due to their routing styles. Bamboo performs lookups recursively, whereas Chord routes iteratively. Chord could easily be changed to route recursively; in fact, newer versions of Chord support both recursive routing and PNS. Note, however, that Chord's latency grows more quickly under increasing churn than does Bamboo's. In Section 3.3.2, we will show evidence to support our belief that this growth is due to Chord's method of choosing timeouts for lookup messages and is independent of the lookup style employed.

Summary

To summarize this section, we note that we have observed several effects of churn on existing DHT implementations. A DHT may fail to complete lookup requests altogether, or it may complete them but return inconsistent results for the same lookup launched from different source nodes. On the other hand, a DHT may continue to return consistent results as churn rates increase, but it may suffer from a dramatic increase in lookup latency in the process.

3.2 The Bamboo DHT

The remainder of this chapter focuses only on the Bamboo DHT, in which we have implemented each alternative design choice studied here. Working entirely within a single implementation allows us to minimize the differences between experiments comparing one design choice to another. We thus complete the the brief description of how Bamboo works from Section 2.1 before continuing.

We built Bamboo after gaining extensive experience implementing Tapestry [ZHS$^+$04], a more sophisticated but also more complicated DHT design. As a result of this experience, our goal in building Bamboo was to produce an extremely simple DHT design on which more advanced functionality could be layered, but which did not depend on such advanced functionality for correct operation. While we built Bamboo on the Pastry geometry, we thus implemented only those features of Pastry that were absolutely necessary to make the system function.

Joining the Network To join an existing Bamboo network, a new node *A* asks an existing node *G* to route a join message to the existing node that is the root for *A*'s identifier. As in Pastry, the nodes that this message traverses in route to the root are recording in the message, and the root responds to the message with all the nodes in this path, its own identifier and network address, plus those of its leaf set. The root also sends an application-level ping to *A*, and if *A* responds, the root adds *A* to its own leaf set.

In contrast to Pastry, which uses the root's response to perform a sophisticated join algorithm designed to maximize the proximity of a node's neighbors, in Bamboo node *A* does very

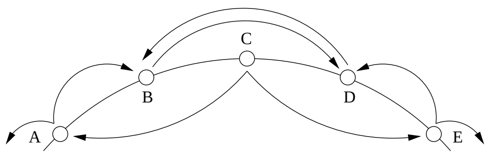


Figure 3.4: *The need for pushing and pulling leaf sets.* Arrows represent neighbor links. Unless leaf sets are also pulled, *C*'s leaf set is never corrected.

little with the result. It simply sends an application-level ping to each node in the response, and if they respond to this ping, it adds them to its leaf set and routing table as appropriate (if they are its immediate predecessors or successors, or if they have the correct prefixes).

Maintaining the Leaf Set To maintain its leaf set over time, a Bamboo node pushes a list of the nodes in its own leaf set to some member of that set, and pulls a list of that neighbor's leaf set in response. In this way, the node learns of new nodes in its vicinity of the ring. This process can be performed periodically or in response to failures, as we describe below.

It is important, however, that a node perform both the push and pull phases. An example is shown in Figure 3.4; indeed, it was observing this kind of state that led us to implement pulls. Five nodes are shown in a system with $\ell = 1$; the arrows represent each node's successor and predecessor according to its leaf set. Node *C* is unavailable during which time *B* and *D* join. *C* subsequently becomes available again, but nodes *B* and *D* have no knowledge of it, whereas *C* still thinks its neighbors are *A* and *E*. If leaf sets are only pushed, no node in this system will tell *C* about the existence of *B* or *D*, and its leaf set will remain incorrect. With pulls, however, the first time *C* contacts *A* it will learn about *B*; the same is true for *E* and *D*.

In the published descriptions of Pastry, nodes only push leaf sets; there does not appear to be a corresponding pull [MCR03]. We are not sure why this is the case.

Maintaining the Routing Table To find a neighbor with prefix *p* for its routing table, a Bamboo node picks an identifier $i = p \cdot s$, where *s* is a random suffix, and does a lookup on *i*. It then sends an application-level ping to the resulting node, and if the node responds, it is added to the routing table. As with leaf set maintenance, this process can be performed periodically or in response to failures.

To find proximal neighbors, a Bamboo node can be configured to repeat this process continually, even for routing table entries for which it already has neighbors. In this latter case, it replaces the existing neighbor if the round-trip time to a newly discovered one is at least 10%

```
public static interface SendCallBack {
   void send_callback (boolean success);
}

public void send (Object msg, InetSocketAddress dst, int tries, SendCallBack cb);

public double est_rtt_ms (InetSocketAddress peer);
```

Figure 3.5: *The Bamboo communications layer interface.* The layer makes up to *tries* attempts to send *msg* to *dst*, calling *send_callback* after an ACK or too many retries. It also exposes the mean observed round-trip time to each peer.

shorter than that to the existing one. Although this process is much less sophisticated than that used by other DHTs to find nearby neighbors, we show in Section 3.3.3 that it is quite effective.

The Messaging Layer Bamboo nodes communicate using UDP. While we originally chose UDP to limit the number of file descriptors used by Bamboo while running multiple virtual nodes on the same machine, we have since come to believe that the semantics of TCP are inappropriate for a DHT. What is needed instead is message-based, unreliable, unordered, but congestion-controlled communication. The manner in which these semantics are provided is briefly described below, but we emphasize here that the specifics should be viewed only as an artifact of the system. In fact, the semantics we desire are quite close to those provided by DCCP [KHFP03] using TCP-like congestion control, and it is likely that we would have used DCCP were it available, although we admit we have not fully explored this possibility.

In the style of TCP, the Bamboo communications layer uses the time between when it sends message and the receipt of the corresponding ACK to maintain an exponentially weighted average round trip time (RTT) and variance thereof for each peer. These values are made available to higher layers of the system. It computes round-trip timeouts (RTOs) to decide when to retransmit a packet, and it backs the RTO off exponentially with each timeout. It maintains a congestion window in a similar manner to the TCP slow-start algorithm, and it notifies the application when a packet is acknowledged. Unlike TCP, our messaging layer does not implement fast retransmit. The interface that the communications layer exports is shown in Figure 3.5.

3.3 Handling Churn

Having given evidence indicating that DHTs' ability to perform lookups is hindered under churn, and having described the details of our Bamboo implementation, we now turn to the heart of this chapter: a study of the factors contributing to DHTs' difficulty with churn, and a comparison of solutions that can be used to overcome them. In turn, we discuss reactive versus periodic recovery from neighbor failure, the calculation of good timeout values for lookup messages, and techniques to achieve proximity in neighbor selection.

3.3.1 Reactive vs. Periodic Recovery

Early implementations of Bamboo suffered performance degradation under churn similar to that of FreePastry. MIT Chord's performance, however, does not degrade in the same way. A significant difference in its behavior is a design choice about how to handle detected node failures. We will call the two alternative approaches reactive and periodic recovery.

Reactive recovery In reactive recovery, a node reacts to the loss of one if its existing leaf set neighbors (or the appearance of a new node that should be added to its leaf set) by sending a copy of its new leaf set to every node in it. To save bandwidth, a node can only send differences from the last message, but the total number of messages is still $O(k^2)$ for a leaf set of k nodes. This algorithm converges quickly, is used in FreePastry, and was the only mode supported in early versions of Bamboo. MSPastry uses a more bandwidth-efficient, but more complex, variant of reactive recovery [CCR03a].

Periodic recovery In contrast, in periodic recovery a node periodically shares its leaf set with each of the members of that set, each of whom responds in kind with its own leaf set. This process takes place independently of the node detecting changes in its leaf set. As a simple optimization, a node picks one random member of its leaf set with which to share state in each period. This change saves bandwidth, but still converges in $O(\log k)$ phases, where k is the size of the leaf set. (Further details can be found elsewhere [RGRK03].) This algorithm is the one currently used by Bamboo, and the periodic nature of this algorithm is shared by Chord's method of keeping its successor list correct.

Positive feedback cycles

Reactive recovery runs the risk of creating a positive feedback cycle as follows. Consider a node whose access link to the network is sufficiently congested such that several consecutive timeouts cause it to believe that one of its neighbors has failed. If the node is recovering reactively, recovery operations begin, and the node will add even more packets to its already congested network link. This added congestion will increase the likelihood that the node will mistakenly conclude that other neighbors have failed. If this process continues, the node will eventually cause congestion collapse on its access link.

Observations of these cycles in the early Bamboo code (and examination of the Chord code) originally led us to propose periodic recovery for handling churn. By decoupling the rate of recovery from the discovery of failures, periodic recovery prevents the feedback cycle described above. Moreover, by lengthening the recovery period with the observation of message timeouts, we can introduce a negative feedback cycle, further improving resilience.

Another way to mitigate the instability associated with reactive recovery is to be more conservative when detecting node failure. We have found one effective approach to be to conclude failure only after 15 consecutive message timeouts to a neighbor. Since timeouts are backed off multiplicatively to a maximum of five seconds, it is unlikely that a node will conclude failure due to congestion. One drawback with this technique, however, is that neighbors that have actually failed remain in a node's routing table for some time. Lookups that would route through these neighbors are thus delayed, resulting in long lookup latencies. To remedy this problem, a node stops routing through a neighbor after seeing five consecutive message timeouts to that neighbor. We have found these changes make reactive recovery feasible for small leaf sets and moderate churn.

Scalability

Experiments show little difference in correctness between periodic and reactive recovery at low churn rates. (At high churn rates, reactive recovery is far worse.) To see why, consider a node *A* that joins a network, and let *B* be the node in the existing network whose identifier most closely matches that of *A*. As in Pastry, *A* retrieves its initial leaf set by contacting *B*, and *B* adds *A* to its leaf set immediately after confirming its IP address and port (with a probe message). Until *A*'s arrival propagates through the network, another node *C* may still route messages that should go to *A* to *B* instead, but *B* will just forward these messages on to *A*. Likewise, should *A* fail, *B* will still be in *C*'s leaf set, so once routing messages to *A* time out, *C* and other nearby nodes will generally

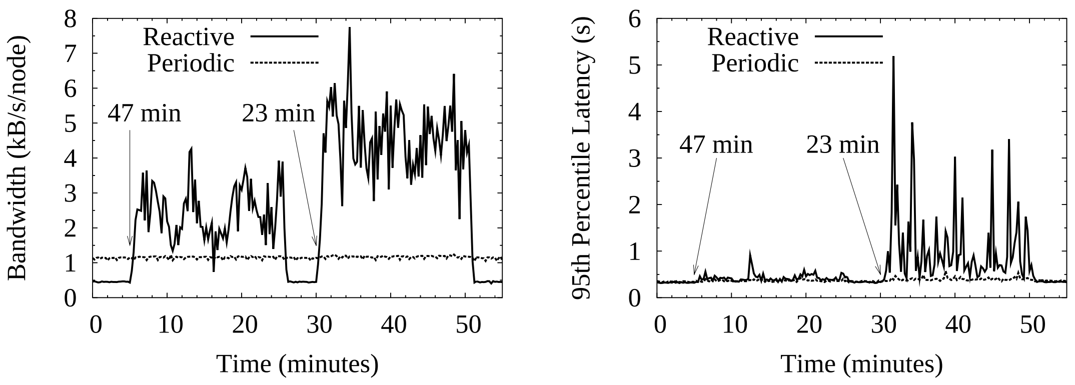


Figure 3.6: *Reactive versus periodic recovery.* Without churn, reactive recovery is very efficient, as messages are only sent in response to actual changes. At reasonable churn rates, however, periodic recovery uses less bandwidth, and lower contention for the network leads to lower latencies.

all agree that *B* is the next best choice.

While both periodic and reactive recovery achieve roughly identical correctness, there is a large difference in the bandwidth consumed under different churn rates and leaf set sizes. (A commonly accepted rule of thumb is that to provide sufficient resilience to massive node failure, the size of a node's leaf set should be logarithmic in the system size.) Under low churn, reactive recovery is very efficient, as messages are only sent in response to actual changes, whereas periodic recovery is wasteful. As churn increases, however, reactive recovery becomes more expensive, and this behavior is exacerbated by increasing leaf set size. Not only does a node see more failures when its leaf set is larger, but the set of other nodes it must notify about the resulting changes in its own leaf set is larger. In contrast, periodic recovery aggregates all changes in each period into a single message.

Figure 3.6 shows this contrast in Bamboo using leaf sets of 24 nodes, the default leaf set size in FreePastry. In this figure, we ran Bamboo using both configurations for two 20-minute periods using 47 and 23 minute median session times, respectively. These two periods were separated by five minutes with no churn.

We note that during the periods of the test where there is no churn, reactive recovery uses less than half of the bandwidth of periodic recovery. On the other hand, under churn its bandwidth use jumps dramatically. As discussed above, Bamboo does not suffer from positive feedback cycles on account of this increased bandwidth usage. Nevertheless, the extra messages sent by reactive recovery compete with lookup messages for the available bandwidth, and as churn increases we see a corresponding increase in lookup latency. Although not shown in the figure, the number of hops per lookup is virtually identical between the two schemes, implying that the growth in bandwidth is most likely due to contention for the available bandwidth.

Since our goal is to handle median session times down to a few minutes with low lookup latency, we do not explore reactive recovery further in this work. The remainder of the Bamboo results we present are all obtained using periodic recovery.

3.3.2 Timeout Calculation

In this section, we discuss the role that lookup message timeouts play in handling churn.

To understand the relative importance of timeouts in a DHT as opposed to a more traditional networked system, consider a traditional client-server system such as the networked file system (NFS). In NFS, the server does not often fail, and when it does there are generally few options for recovery and no alternative servers to fail over to. If a response to an NFS request is not received in the expected time, the client can only try again with an exponentially increasing timeout value.

In a peer-to-peer system under churn, in contrast, requests will be frequently sent to a node that has left the system, possibly forever. At the same time, a DHT has many alternate paths available to complete a lookup. Simply backing off the request period is thus a poor response to a request timeout; it is often better to retry the request through a different neighbor.

A node should ensure that the timeout for a request was judiciously selected before routing to an alternate neighbor. If it is too short, the node to which the original was sent may be yet to receive it, may be still processing it, or the response may be queued in the network. If so, injecting additional requests may result in the use of additional bandwidth without any beneficial result—for example, in the case that the local node's access link is congested. Conversely, if the timeout is too long, the requesting node may waste time waiting for a response from a node that has left the network. If the request rate is high, these long waits may cause unbounded queue growth on the requesting node that might be avoided with shorter timeouts.

For these reasons, nodes should accurately choose timeouts such that a late response is indicative of node failure, rather than network congestion or processor load.

Techniques

We discuss and study three alternative timeout calculation strategies. In the first, we fix all timeouts at a conservative value of five seconds as a control. In the second, we calculate TCP-style timeouts using direct measurement of past response times. Finally, we explore using indirect measurements from a virtual coordinate algorithm to calculate timeouts.

TCP-style timeouts: If a DHT routes recursively, it rarely communicates with nodes other than its direct neighbors in the overlay network. Since the number of these neighbors is logarithmic in the size of the network, and since each node periodically probes each neighbor for availability, a node can easily maintain a past history of each neighbor's response times for use in calculating timeouts. In Bamboo, we have implemented this strategy following the style of the early TCP work [JK88], where each node maintains an exponentially weighted mean and variance of the response time for each neighbor. Specifically, the estimate round-trip timeout (RTO) for a neighbor is calculated as

$$\mathrm{RTO} = \mathrm{AVG} + 4 \times \mathrm{VAR},$$

where AVG is the observed average round-trip time and VAR is the observed mean variance of that time.

Timeouts from virtual coordinates: In contrast to recursive routing, with iterative routing a node must potentially have a good timeout for *any* other node in the network. However, in some scenarios iterative routing does have attractive properties. For example, since the source of a lookup request controls the entire process of iterative routing, it is easy to explore several different lookup paths in parallel. For only a constant increase in bandwidth used, this technique prevents a single timeout from delaying a lookup [LSG$^+$04].

Virtual coordinates provide one approach to computing timeouts without previously measuring the response time to every node in the system. In this scheme, a distributed machine learning algorithm is employed to assign to each node coordinates in a virtual metric space such that the distance between two nodes in the space is proportional to their latency in the underlying network.

Bamboo includes an implementation of the Vivaldi coordinate system employed by Chord [CDK$^+$03b]. Vivaldi keeps an exponentially-weighted average of the error of past round-trip times calculated with the coordinates, and computes the RTO as

$$\mathrm{RTO} = v + 6 \times \alpha + 15$$

where v is the predicted round-trip time and α is the average error. The constant term of 15 milliseconds is added to avoid unnecessary retransmissions when the destination is the local host.

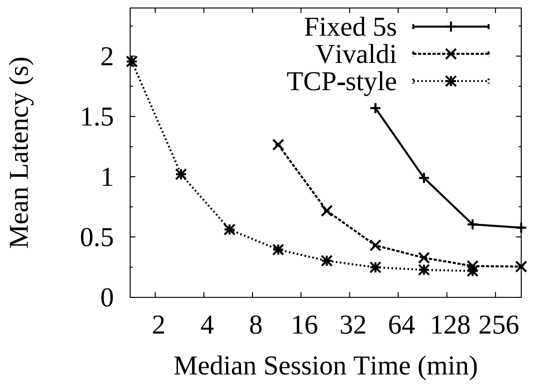


Figure 3.7: *TCP-style versus virtual coordinate-based timeouts in Bamboo.* Timeouts chosen using Vivaldi are competitive with TCP-style timeouts for moderate churn rates.

Results

TCP-style timeouts assume a recursive routing algorithm, and a virtual coordinate system is necessary only when routing iteratively. While we would ideally compare the two approaches by measuring each in its intended environment, this would prevent us from isolating the effect of timeouts from the differences caused by routing styles.

Instead, we study both schemes under recursive routing. If timeouts calculated with virtual coordinates provide performance comparable to those calculated in the TCP-style under recursive routing, we can expect the virtual coordinate scheme to not be prohibitively expensive under iterative routing. While other issues may remain with iterative routing under churn (e.g. congestion control—see Section 3.5), this result would be a useful one.

Figure 3.7 shows a direct comparison of the three timeout calculation methods under increasing levels of churn. In all cases in this experiment, the Bamboo configurations differed only in choice of timeout calculation method. Proximity neighbor selection was used, but the latency measurements for PNS used separate direct probing and not the virtual coordinates.

Even under light levels of churn, fixing all timeouts at five seconds causes lookup timeouts to pull the mean latency up to more than twice that of the other configurations, confirming our intuition about the importance of good timeout values in DHT routing under churn. Moreover, by comparing Figure 3.7 to Figure 3.3, we note that under high churn timeout calculation has a greater effect on lookup latency than the use of PNS.

Virtual coordinate-based timeouts achieve very similar mean latency to TCP-style timeouts at low churn. Furthermore, they perform within a factor of two of TCP-style measurements until the median churn rate drops to 23 minutes. Past this point, their performance quickly diverges, but virtual coordinates continue to provide mean lookup latencies under two seconds down to twelve-minute median session times.

Finally, we note the similarity in shape of Figure 3.7 to Figure 3.3, where we compared the performance of Chord to Bamboo, suggesting that the growth in lookup latency under Chord at high churn rates is due to timeout calculation based on virtual coordinates.

3.3.3 Proximity Neighbor Selection

Perhaps one of the most studied aspects of DHT design has been proximity neighbor selection (PNS), the process of choosing among the potential neighbors for any given routing table entry according to their network latency to the choosing node. This research is well motivated. The *stretch* of a lookup operation is defined as the latency of the lookup divided by the round-trip time between the lookup source and the node discovered by the lookup in the underlying IP network. Dabek et al. present an argument and experimental data that suggest that PNS allows a DHT of N nodes to achieve median stretch of only 1.5, independent of the size of the network and despite using $O(\log N)$ hops [DLS+04]. Others have proved that PNS can be used to provide constant stretch in locating replicas under a restricted network model [PRR97]. This is the first study of which we are aware, however, to compare methods of achieving PNS under churn. We first take a moment to discuss the common philosophy and techniques shared by each of the algorithms we study.

Commonalities

One of the earliest insights in DHT design was the separation of correctness and performance in the distinction between neighbors in the leaf set and neighbors in the routing table [RD01, SMK+01]. So long as the leaf sets in the system are correct, lookups will always return correct results, although they may take $O(N)$ hops to do so. Leaf set maintenance is thus given priority over routing table maintenance by most DHTs. In the same manner, we note that so long as each entry in the routing table has *some* appropriate neighbor (i.e., one with the correct identifier prefix), lookups will always complete in $O(\log N)$ hops, even though they make take longer than if the neighbors had been chosen for proximity. We say such lookups are *efficient*, even though they may not have low stretch. By this argument, we reason that it is desirable to fill a routing table entry quickly, even with a less than optimal neighbor; finding a nearby neighbor is a lower priority.

There is a further argument to treating proximity as a lower priority in the presence of churn. Since we expect our set of neighbors to change over time as part of the churn process, it makes little sense to work too hard to find the absolute closest neighbor at any given time; we might expend considerable bandwidth to find them only to see them leave the network shortly afterward. As such, our general approach is to run each of the algorithms described below *periodically*. In the case where churn is high, this technique allows us to retune the routing table as the network changes. When churn is low, rerunning the algorithms makes up for latency measurement errors caused by transient network conditions in previous runs.

Our general approach to finding nearby neighbors thus takes the following form. First, we use one of the algorithms below to find nodes that may be near to the local node. Next, we measure the latency to those nodes. If we have no existing neighbor in the routing table entry that the measured node would fill, or if it is closer than the existing routing table entry, we replace that entry, otherwise we leave the routing table unchanged. Although the bandwidth cost of multiple measurements is high, the storage cost to remember past measurements is low. As a compromise, we perform only a single latency measurement to each discovered node during any particular run of an algorithm, but we keep an exponentially weighted average of past measurements for each node, and we use this average value in deciding the relative closeness of nodes. This average occupies only eight bytes of memory for each measured node, so we expect this approach to scale comfortably to very large systems.

Techniques

The techniques for proximity neighbor selection that we study here are global sampling, sampling of our neighbors' neighbors, and sampling of the nodes that have our neighbors as their neighbors. We describe each of these techniques in turn.

Global sampling In global sampling we use the lookup functionality of the DHT to find new neighbors. For a routing table entry that requires a neighbor with prefix p, we perform a lookup for a random identifier with prefix p. The node returned by this lookup will almost always have the desired prefix. (As an example of why this is not always the case, note that a lookup of identifier 0 may return a node whose identifier starts with 1 if the node with the largest identifier in the ring is numerically closer to 0 than the node with the smallest identifier.) Given enough samples, all nodes with prefix p will eventually be probed. The motivation for this technique comes from Gummadi et al., who showed that sampling only around 16 nodes for each routing table entry provides almost

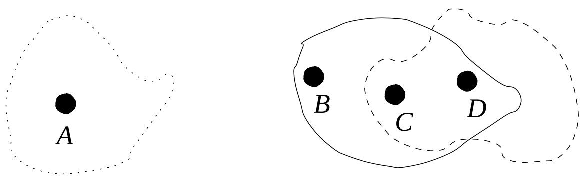


Figure 3.8: *Sampling neighbors' neighbors.* If *A* joins using *D* as its gateway, its initial level-0 neighbors are the same as those of *D*; assume that these are all within the dashed line. *A* contacts a level-0 neighbor, e.g. *C*, and asks it for its level-0 neighbors. *A* would learn about *B* in this manner. However, there may be no path from the *D*'s ideal neighbors to those of *A*.

optimal proximity [GGG$^+$03].

There are some cases, however, where global sampling will take unreasonably long to find the closest possible neighbor. For example, consider two nodes separated from the core Internet by the same, high latency access link, as shown in Figure 3.9. The relatively high latency seen by these two nodes to all other nodes in the network makes them attractive neighbors for each other; if they have different first digits in a network with logarithm base two, they can drastically reduce the cost of the first hop of many routes by learning about each other. However, the time for these nodes to find each other using global sampling is proportional to the size of the total network, and so they may not find each other before their sessions end. It is this drawback of global sampling that leads us to consider other techniques.

Neighbors' neighbors The next technique we consider is sampling our neighbors neighbors, a process called *routing table maintenance* in the Pastry work [RD01]. In this technique, we contact an existing routing table neighbor at level l of our routing table and ask for its level l neighbors. Like us, these nodes share a prefix of $l-1$ digits with the contacted neighbor and are thus appropriate for use in our routing table as well. As in global sampling, having discovered these new nodes, we probe them for latency and use them if they are closer than our existing neighbors.

The motivation for sampling neighbors' neighbors is illustrated in Figure 3.8; it relies on the expectation that proximity in the network is roughly transitive. If a node discovers one nearby node, then that node's neighbors are probably also nearby. In this way, we expect that a node can "walk" through the graph of neighbor links to the set of nodes most near it.

To see one possible shortcoming of sampling our neighbors' neighbors, consider again Figure 3.9. While the two isolated nodes would like to discover each other, it is unlikely that any other nodes in the network would prefer them as neighbors; their isolation makes them unattractive for routing lookups that originate elsewhere, except in the rare case that they are the result of those

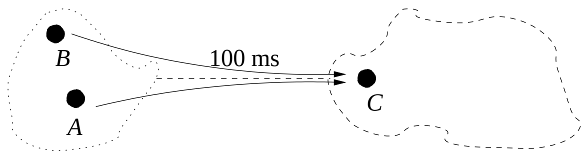


Figure 3.9: *Sampling neighbors' inverse neighbors.* Nodes *A* and *B* are isolated from the remainder of the network by a long latency, and are initially unaware of each other. Such a situation is possible if, for example, two European nodes join a network of primarily North American nodes. As such, they make unattractive neighbors for other nodes, but they would still like to find each other. If they both have *C* as a neighbor, they can find each other by asking *C* for its inverse neighbors.

lookups. As such, since neighbor links in DHTs are rarely symmetric, it is unlikely that there is a path through the graph of neighbor links that will lead one isolated node to the other, despite their relative proximity.

Neighbors' inverse neighbors The latter argument presents an obvious alternative approach. Instead of sampling our neighbors' neighbors, why not sample those nodes that have the same neighbors as the local node? This technique was originally proposed in the Tapestry nearest neighbor algorithm [HKRZ02]; we call it sampling our neighbors' inverse neighbors. To motivate this technique, consider again Figure 3.9. Although the two remote nodes are unlikely to be neighbors of many other nodes, given that their existing neighbors are mostly nearby, they are likely to choose the same neighbors from outside their isolated domain. For this reason, they are likely to find each other in the set of their neighbors' inverse neighbors.

Normally, a DHT node would not record the set of nodes that use it as a neighbor. Actively managing such a list, in fact, requires additional probing bandwidth. Currently, the Bamboo implementation does actively manage this set, but it could be easily approximated at each node by keeping track of the set of nodes that have sent it liveness probes in the last minute or so. We plan to implement this optimization in our future work.

Recursive sampling Consider Figure 3.9 one final time, and assume that we are using a single-bit digits and that the two remote nodes begin with different digits, i.e. 0 and 1 respectively. The node whose identifier starts with 0 will have only one neighbor whose identifier begins with 1 (its level-0 neighbor). Likewise, the node whose identifier starts with 1 will have only one neighbor that starts with 0. The set of neighbors in whose inverse neighbor sets the two isolated neighbors can find each other is thus very small. As such, until the two isolated nodes have found very nearby level-0

```
(1) function nearestNeighbors () =
(2)     S = highestNonempRtLevel ();
(3)     l = longestMatchingPrefix (S);
(4)     while l >= 0
(5)         forall n in S
(6)             T = n.getInverseRtNeighbors (l);
(7)             S = closest (k, S ∪ T);
```

Figure 3.10: *The Tapestry nearest neighbor algorithm.*

neighbors, they will be unlikely to find each other among their neighbors' inverse neighbors.

To remedy this final problem, we can perform the sampling of nodes in a manner similar to that used by the Tapestry nearest neighbor algorithm and the Pastry optimized join algorithm. Pseudo-code for this technique is shown in Figure 3.10. Starting with the highest level l in its routing table, a node contacts the neighbors at that level and retrieves their neighbors or inverse neighbors. The latency to each newly discovered node is measured, and all but the k closest are discarded. The node then decrements l and retrieves the level-l neighbors from each non-discarded node. This process is repeated until $l < 0$. Along the way, each discovered neighbor is considered as a candidate for use in the routing table. To keep the cost of this algorithm low, we limit it to having at most three outstanding messages (neighbor requests or latency probes) at any time.

Note that although this process starts by sampling from the routing table, the set of nodes on which it recurses is not constrained by the prefix-matching structure of that table. As such, it does not suffer from the small rendezvous set problem discussed above. In fact, under certain network assumptions, it has been proved that this process finds a node's nearest neighbor in the underlying network.

Results

In order to compare the techniques described above, it is important to consider not only effective they are at finding nearby neighbors, but also at what bandwidth cost they do so. For example, global sampling at a high enough rate relative to the churn rate would achieve perfect proximity, but at the cost of a very large number of lookups and latency probes. To make this comparison, then, we ran each algorithm (and some combinations of them) at various periods, then plotted the mean lookup latency under churn versus bandwidth used. The results for median session

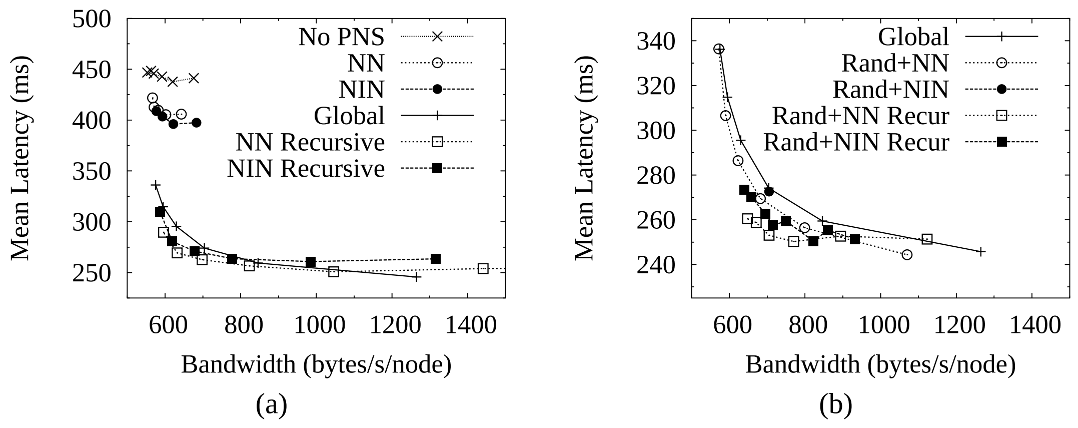


Figure 3.11: *Comparison of PNS techniques.* "No PNS" is the control case, where proximity is ignored. "Global Sampling" uses the lookup function to sample all nodes in the DHT. "NN" is sampling our neighbor's neighbors, and "NIN" is sampling their inverse neighbors. The recursive versions of "NN" and "NIN" mimic the nearest-neighbor algorithms of Pastry and Tapestry, respectively. Note that the scales are different between the two figures.

times of 47 minutes are shown in Figure 3.11, which is split into two graphs for clarity.

Figure 3.11(a) shows several interesting results. First, we note that only a little bit of global sampling is necessary to produce a drastic improvement in latency versus the configuration that is not using PNS. With virtually no increase in bandwidth, global sampling drops the mean latency from 450 ms to 340 ms.

Next, much to our surprise, we find that simple sampling of our neighbor's neighbors or inverse neighbors is not terribly effective. As we argued above, this result may be in part due to the constraints of the routing table, but we did not expect the effect to be so dramatic. On the other hand, the recursive versions of both algorithms are at least as effective as global sampling, but not much more so. This result agrees with the contention of Gummadi et al. that only a small amount of global sampling is necessary to achieve near-optimal PNS.

Figure 3.11(b) shows several combinations of the various algorithms. Global sampling plus sampling of neighbors' neighbors does well, offering a small decrease in latency without much additional bandwidth. However, the other combinations offer similar results. At this point, it seems prudent to say that the most effective technique is to combine global sampling with any other technique. While there may be other differences between the techniques not revealed by this analysis, we see no clear reason to prefer one over another as yet.

3.4 Related Work

As we noted at the start of this chapter, while DHTs have been the subject of much research in the last 4 years or so, there have been few studies of the churn resilience of real implementations at scale, perhaps because of the difficulty of deploying, instrumenting, and creating workloads for such deployments. However, there has been a substantial amount of theoretical and simulation-based work.

Gummadi et al. [GGG$^+$03] present a comprehensive analysis of the resilience of the various DHT geometries to failures.

Liben-Nowell et al. [LNBK02] present a theoretical analysis of structured peer-to-peer overlays from the point of view of churn as a continuous process. They prove a lower bound on the maintenance traffic needed to keep such networks consistent under churn, and show that Chord's algorithms are within a logarithmic factor of this bound. This chapter, in contrast, has focused more on the systems issues that arise in handling churn in a DHT. For example, we have observed what they call "false suspicions of failure", the appearance that a functioning node has failed, and shown how reactive failure recovery can exacerbate such conditions.

Mahajan et al. [MCR03] present a simulation-based analysis of Pastry in which they study the probability that a DHT node will forward a lookup message to a failed node as a function of the rate of maintenance traffic. They also present an algorithm for automatically tuning the maintenance rate for a given failure rate. Since this algorithm increases the rate of maintenance traffic in response to losses, we are concerned that it may cause positive feedback cycles like those we have observed in reactive recovery. Moreover, we believe their failure model is pessimistic, as they do not consider hop-by-hop retransmissions of lookup messages. By acknowledging lookup messages on each hop, a DHT can route around failed nodes in the middle of a lookup path, and in this work we have shown that good timeout values can be computed to minimize the cost of such retransmissions.

Castro et al. [CCR03a] presented a number of optimizations they have performed in MSPastry, the Microsoft Research implementation of Pastry, using simulations. Also, Li et al. [LSM$^+$05, LSG$^+$04] performed a detailed simulation-based analysis of several different DHTs under churn, varying their parameters to explore the latency-bandwidth tradeoffs presented. It was their work that inspired our analysis of different PNS techniques.

As opposed to the emulated network used in this study, simulations do not usually consider such network issues as queuing, packet loss, etc. By not doing so, they either allow simulation of far larger networks than we have studied here [CCR03a, MCR03], or they are able to explore a far larger space of possible DHT configurations [LSM+05, LSG+04]. On the other hand, they do not reveal subtle issues in DHT design, such as the tradeoffs between reactive and periodic recovery. Also, they do not reveal the interactions of lookup traffic and maintenance traffic in competing for network bandwidth. We are interested in whether a useful middle ground exists between these approaches.

Finally, a number of useful features for handling churn have been proposed, but are not implemented by Bamboo. For example, Kademlia [MM02] maintains several neighbors for each routing table entry, ordered by the length of time they have been neighbors. Newer nodes replace existing neighbors only after failure of the latter. This design decision is aimed at mitigating the effects of the high "infant mortality" observed in peer-to-peer networks.

Another approach to handling churn is to introduce a hierarchy into the system, through stable "superpeers" [gnu, ZDH+02]. While an explicit hierarchy is a viable strategy for handling load in some cases, this work has shown that a fully decentralized, non-hierarchical DHT can in fact handle high rates of churn at the routing layer.

3.5 Future Work

As discussed in the introduction, there are several other limitations of this study that we think provide for important future work. At an algorithmic level, we would like to study the effects of alternate routing table neighbors as in Kademlia and Tapestry. We would also like to continue our study of iterative versus recursive routing. While we show in Chapter 7 that iterative routing has performance advantages in networks with a significant fraction of slow nodes, we have yet to study it under heavy churn. Furthermore, congestion control for iterative lookups is a challenging problem. The Chord DHT uses a congestion control algorithm [DLS+04] called STP for this purpose, but its behavior under churn has not been tested either.

At a methodological level, we would like to broaden our study to include better models of network topology and churn. We have so far used only a single network topology in our work, and so our results should be not be taken as the last word on PNS. In particular, the distribution of internode latencies in our ModelNet topology is more Gaussian than the distribution of latencies measured on the Internet. Unfortunately for our purposes, these measured latency distributions do not include topology information, and thus cannot be used to simulate the kind of network cross traffic that we have found important in this study. The existence of better topologies would be most welcome. While we have been running our code successfully on a real network—PlanetLab—for over a year, PlanetLab is difficult to use for the sort of controlled churn experiments we describe here.

In addition to more realistic network models, we would also like to include more realistic models of churn in our future work. One idea that was suggested to us by an anonymous reviewer was to scale traces of session times collected from deployed networks to produce a range of churn rates with a more realistic distribution. We would like to explore this approach. Nevertheless, we believe that the effects of the factors we have studied are dramatic enough that they will remain important even as our models improve.

Finally, in this work we have only shown the resistance of the Bamboo lookup layer to churn. As noted in Chapter 2, the ability of a DHT to handle churn at the storage layer is limited by the available bandwidth, and we do not expect a DHT storing large amounts of data to handle the degree of churn we have studied in this chapter. That said, we do show in Chapter 5 that OpenDHT can handle the churn on PlanetLab, and we would like to study the resilience of other DHT-based primitives—such as multicast—to churn in the future.

3.6 Conclusion

In this chapter we have summarized the rates of churn observed in deployed peer-to-peer systems and shown that existing DHTs exhibit less than desirable performance at the higher end of these churn rates. We have presented Bamboo and explored various design tradeoffs and their effects on its ability to handle churn.

The design tradeoffs we studied in this chapter fall into three broad categories: reactive versus periodic recovery from neighbor failure, the calculation of timeouts on lookup messages, and proximity neighbor selection. We have presented the danger of positive feedback cycles in reactive recovery and discussed two ways to break such cycles. First, we can make the DHT much more cautious about declaring neighbors failed, in order to limit the possibility that we will be tricked into recovering a non-faulty node by network congestion. Second, we presented the technique of periodic recovery. Finally, we demonstrated that reactive recovery is less efficient than periodic recovery under reasonable churn rates when leaf sets are large, as they would be in a large system.

With respect to timeout calculation, we have shown that TCP-style timeout calculation performs best, but argued that it is only appropriate for lookups performed recursively. It has long been known that recursive routing provides lower latency lookups than iterative, but this result presents a further argument for recursive routing where the lowest latency is important. However, we have also shown that while they are not as effective as TCP-style timeouts, timeouts based on virtual coordinates are quite reasonable under moderate rates of churn. This result indicates that at least with respect to timeouts, iterative routing should not be infeasible under moderate churn. Moreover, as suggested by the results in Chapter 7, the ease with which lookup can be parallelized under iterative routing may afford additional resilience to churn.

Concerning proximity neighbor selection, we have shown that global sampling can provide a 24% reduction in latency for virtually no increase in bandwidth used. By using an additional 40% more bandwidth, a 42% decrease in latency can be achieved. Other techniques are also effective, especially our adaptations of the Pastry and Tapestry nearest-neighbor algorithms, but not much more so than simple global sampling. Merely sampling our neighbors' neighbors or inverse neighbors is not very effective in comparison. Some combination of global sampling an any of the other techniques seems to provide the best performance at the least cost.

Chapter 4

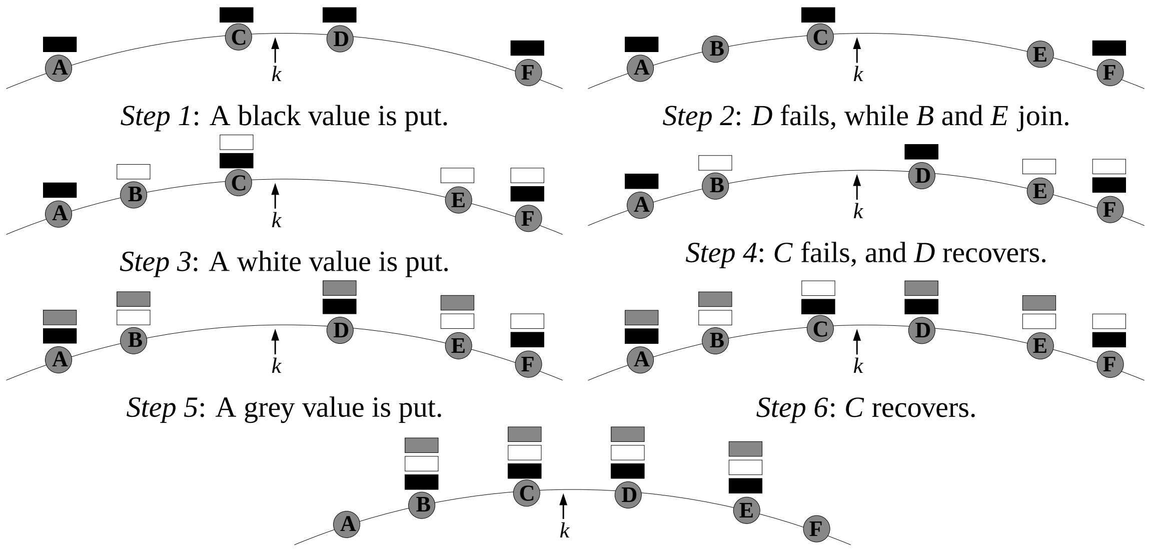
Storage

In the previous chapter we explored how to implement DHT lookup in a churn-resilient manner. While it is important as a building block, the lookup interface is too low-level for many DHT applications, which instead require higher-level functionality such as put/get or DOLR. As discussed in Chapter 1, these interfaces share a common piece of functionality: fault-tolerant storage of key-value pairs. While some applications can implement their own fault-tolerant storage using the soft-state approach described in Chapter 2, for many others fault-tolerance is much more efficiently supported by the DHT itself. This chapter describes Bamboo's fault-tolerance storage layer.

4.1 Background

As described in the introduction, the goal of Bamboo's storage algorithm is to store a given a key-value pair (k, v) on pred$_i(k)$ and succ$_i(k)$ for $i \in [1, \ell']$, where $\ell' \leq \ell$, the leaf set radius. To simplify the remainder, we introduce the following terms. We denote by r the number of replicas for each value; i.e., $r = 2\ell'$. We call the set of Bamboo nodes that store values under key k the *replica set* for k, and we denote this set $R(k)$. We refer to members of this set as replicas for k when the meaning is clear. Furthermore, we denote by $R^{-1}(A)$ the set of keys for which a node A stores replicas, and we say that A is *responsible* for the keys in $R^{-1}(A)$.

To see why this problem is so challenging, consider Figure 4.1, which illustrates the process of storing three values with the same key onto six intermittently-available DHT nodes. Initially, $R(k) = \{A, C, D, F\}$, so when the black value is put, it is stored by nodes A, C, D, and F. Then node D fails, and nodes B and E join, changing $R(k)$ to $\{B, C, E, F\}$. Next, a white value is put. Then node C fails, and node D recovers, changing $R(k)$ to $\{A, B, D, E\}$. Then, a grey value is



Step 1: A black value is put.
Step 2: *D* fails, while *B* and *E* join.
Step 3: A white value is put.
Step 4: *C* fails, and *D* recovers.
Step 5: A grey value is put.
Step 6: *C* recovers.
Step 7: The desired final state.

Figure 4.1: *The storage problem.* Values are put into the DHT as nodes fail and recover. The goal of the storage algorithm is to reach step 7 from step 6, even though no node knows all the values.

put. Finally, node C recovers, changing $R(k)$ to $\{B,C,D,E\}$. While all values were stored onto the correct set of nodes at the time they were put into the system, by Step 6 in the figure, they are no longer all stored at the correct nodes. Further complicating the problem, no one node is storing all three values.

Our initial approach to solving this problem was to have the root for each key keep track of nodes as they joined and left its leaf set and thereby determine when a value was no longer sufficiently replicated and on what nodes new replicas should be created. Although this technique seems relatively straightforward, there are a number of tricky corner cases in implementing it, and we were never able to complete a working implementation in Bamboo.

Our next thought towards solving the problem was that rather than carefully tracking all node joins and departures, we should instead implement a continuous process in which each misplaced value “migrated” towards its correct set of replicas. In working out the details of such a process, we we unknowingly re-invented simple epidemic techniques. Shortly thereafter, we rediscovered the epidemic literature (e.g. [DGH$^+$87, JT75, VvRB02]). Because many of the techniques we will explore later in this chapter are in fact special cases of a broader spectrum of epidemic algorithms, we present a brief overview of these algorithms before continuing.

4.2 Introduction to Epidemic Algorithms

In this section we briefly describe two classes of epidemic algorithms, *anti-entropy* and *rumor mongering*. We follow the terminology of Demers et al. [DGH+87], who present a comprehensive overview of the field.

4.2.1 Anti-Entropy

Consider a set R of r replicas for a database. For the moment, let us assume that once a key-value tuple is written to this database, it cannot be changed, so that the consistency problem is reduced to ensuring that every replica contains every tuple. This situation corresponds to the the set of Bamboo nodes in $R(k)$ trying to ensure that they each have all values put under k, for example.

Anti-entropy is a procedure run periodically by each replica whereby it contacts a remote replica and synchronizes its database with that peer, transferring each tuple known only to one replica to the other. When peers are chosen uniformly and randomly, this technique is known to propagate a new tuple to every replica in $O(\log r)$ periods [Pit87].

There are two basic variations of anti-entropy: *push* and *pull*. In push, every period a replica A chooses a random peer B and sends B all the values that A is currently storing. In pull, A chooses a random peer B and asks it for all the values that B is currently storing.

While both push and pull anti-entropy converge in $O(\log r)$ periods, the constants are different. In particular, pull is more efficient when most replicas already have a value; since the probability of contacting a random replica that does not know the value is small, a replica that does not know the value is likely to contact one that does. However, if only one replica has a value, a replica using pull that does not have the value is unlikely to chose the one other replica that does. In contrast, push more efficiently moves a value stored by only one replica to the rest, since early in the process each peer this one replica chooses is unlikely to already have the value.

4.2.2 Rumor Mongering

Consider again our set of replicas, and consider a tuple that originally exists in only one replica. Rumor mongering proceeds as follows: as soon as a replica learns about a new tuple, it starts sending it to other replicas. Each time it sends the rumor to a replica that is already aware of it, however, it will stop spreading the rumor with probability $1/p$. This process begins with the first replica and continues until every replica that learns about the tuple has given up spreading it.

Unlike anti-entropy, rumor mongering does not guarantee that every replica receives the tuple.[1] However, rumor-mongering is much more efficient at propagating a previously unknown value to a set of replicas for low cost. For example, when $p = 1$, the expected fraction of replicas that receive the tuple is 0.8, while the expected number of messages sent is only $1.74r$ [DGH$^+$87].

4.2.3 Epidemic Algorithms and System Stability

We close this section with a note concerning the stability of epidemic algorithms. Vogels, van Renesse, and Birman [VvRB02] note a condition in many group communication technologies that they compare to thrashing in an operating system: in trying to recover from an apparent host failure, the system often causes further network stress, which itself is often mistaken for the failure of other hosts, resulting in a positive feedback cycle. This behavior is similar to that we observed in FreePastry in Chapter 3.

In contrast, with epidemic algorithms, we are always able to move the system from an incorrect state to a "more correct" one; each round of anti-entropy or rumor mongering has the potential to improve the state of the system, and this potential is largely independent of the period between rounds. For a given level of sustainable network stress, then, we can choose a period to avoid ever overloading the network. Given sufficient load, even this throttling will not save the system, but it does prevent the network overload that would otherwise occur with even lower stress levels. As another parallel to Chapter 3, one can consider the leaf set maintenance algorithm of Bamboo as a simple form of anti-entropy; the database being shared is just a list of the nodes whose identifiers lie in a local area of the ring.

4.3 Epidemic Algorithms Meet DHTs

We now describe Bamboo's storage subsystem's algorithms and show their relationship to epidemic algorithms.

4.3.1 Replica Synchronization

In replica synchronization, members of a replica set contact each other and pull values from their peers. Figure 4.2 illustrates this process: a replica A periodically chooses a random replica $B \neq A$, computes the set of keys for which they are both responsible, $R^{-1}(A) \cap R^{-1}(B)$, and

[1]We note that the probability of this failure case can be driven arbitrarily low; when rumors are passed to recipients chosen uniformly and randomly, the fraction of replicas that do not receive a tuple is exponentially small in p.

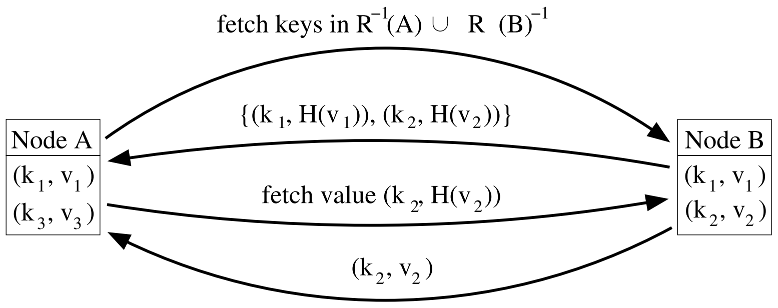

fetch keys in R⁻¹(A) ∪ R (B)⁻¹
{(k 1, H(v 1)), (k 2, H(v 2))}
fetch value (k 2, H(v 2))
(k 2, v 2)
Node A
(k 1, v 1)
(k 3, v 3)
Node B
(k 1, v 1)
(k 2, v 2)

Figure 4.2: *Replica synchronization.* Node A asks node B which key-value pairs it is storing in their common range. Node B replies with a $(k,H(v))$ pair for each key and value it is storing, where $H(v)$ is the SHA-1 hash of v. A compares these pairs with those it is storing, and then asks B for the pair corresponding to $(k_2,H(v_2))$, which A does not recognize. Finally, B sends (k_2,v_2) to A. Note that B need not know all the pairs for this procedure to work. Also, B will learn about (k_3,v_3) when it initiates replica synchronization with A.

performs anti-entropy with B over this set. Specifically, for each key-value pair (k,v) that B has stored, it will send A the pair $(k,H(v))$, where $H(v)$ is the SHA-1 hash of v. Each node indexes the values it has stored by their keys and SHA-1 hashes, and A uses B's response to determine whether B has any values it does not. If so, A pulls those values from B.

Basic properties of replica synchronization We note that replica synchronization is just simple anti-entropy. Most importantly, it is easy to see that it is correct in the following sense: given that at least one replica in $R(k)$ has a value, all replicas in $R(k)$ will eventually obtain it. It follows from the epidemic literature that replica synchronization is also efficient in the sense that the value will be fully replicated in a number of periods logarithmic in the number of replicas. Furthermore, since we expect the replicas in $R(k)$ to be mostly consistent most of the time, we see from the literature that pull is a better approach than push for replica synchronization.

As currently specified, however, the bandwidth cost of replica synchronization is proportional to the number of values stored on each replica; in each synchronization between A and B, B sends A one $(k,H(v))$ pair for every value B stores. In the remainder of this section we explore ways to reduce this cost.

Using Merkle trees as summaries As pictured in Figure 4.3, a Merkle tree [Mer88] is a data structure that allows one to check the integrity of an m-byte subrange of a n-byte file using only $O(m\log n)$ state. In the bottom row of the figure, $d_1,d_2,\ldots$ represent the blocks of a file. Each interior node of the tree is the secure hash of its two children; e.g., $h_6 = h(h_3 \cdot h_4)$, where $\cdot$ represents

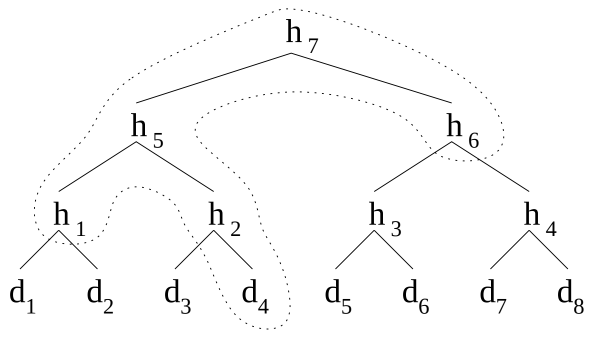


Figure 4.3: *A Merkle tree.* If we already know the root of the tree and its height, we can verify the integrity of block d_3 using only the data inside the dotted line.

concatenation.

Assume that a node A knows the root of the tree, h_7, and its height, and another node B sends it d_3 and all of the information contained inside the dotted line in the figure. Node A can check the integrity of the d_3 sent by B as follows. By the one-wayness of the secure hash, it is presumably hard to find another h_5 or h_6 such that $h_7 = h(h_5 \cdot h_6)$. So by checking that the root it already knows is in fact equal to $h(h_5 \cdot h_6)$, A verifies the integrity of h_5. Likewise, it is hard to find another h_1 or h_2 such that $h_5 = h(h_1 \cdot h_2)$, and in this way A verifies the integrity of h_2. Finally, A uses $h_2 = (d_3 \cdot d_4)$ to verify the integrity of d_3.

Using Merkle trees in anti-entropy We can use Merkle trees to reduce the cost of anti-entropy between two hosts A and B as follows: we sort the $(k, H(v))$ pairs for all $k \in R^{-1}(A) \cap R^{-1}(B)$ and build a Merkle tree above them. The anti-entropy process between two hosts A and B then proceeds as shown in Figure 4.4. First, A sends B the SHA-1 hash of the root and the height of its tree. If this root and height are the same as B's, then the two hosts have the same set of $(k, H(v))$ pairs, so B sends back a success message and the process is complete. Otherwise, B sends back its root's children. Next, A compares each of its own root's children to B's; on each hash where the two differ, A recurses, fetching the associated child blocks from B. This process continues until the leaf (data) blocks are reached, after which A will have fetched from B any $(k, H(v))$ pairs for values B has that A does not. Finally, A can fetch these values from B, and the process is complete.

We make two observations about this process. First, if the Merkle trees for A and B differ only along the right-hand side, then the cost to discover their differences is $O(d \log n)$, where d is the number of values in which they differ and n is the total number of values they store. Compared to the cost of the algorithm without Merkle trees, $O(n)$, this is a marked improvement for mostly-

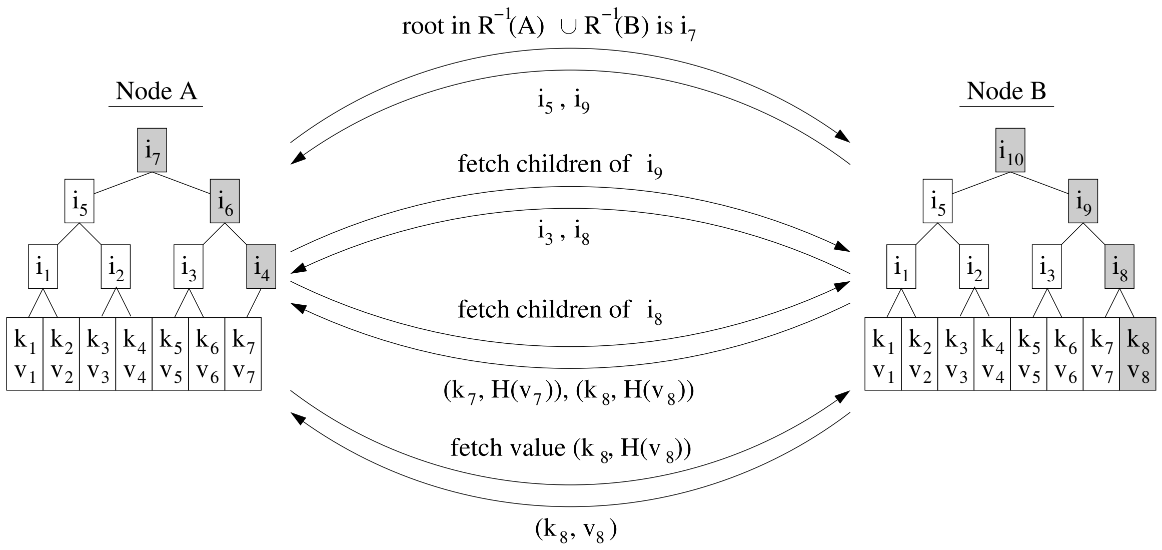

root in $R^{-1}(A) \cup R^{-1}(B)$ is i_7
Node A
Node B
i_5 , i_9
fetch children of i_9
i_3 , i_8
fetch children of i_8
$(k_7, H(v_7)), (k_8, H(v_8))$
fetch value $(k_8, H(v_8))$
(k_8, v_8)
i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_{10}
k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6 k_7 v_7 k_8 v_8

Figure 4.4: *Replica synchronization with Merkle trees.* Nodes *A* and *B* differ in the shaded blocks. Node *A* asks *B* for the root of the Merkle tree that covers their shared range. *B* responds, and *A* asks for the children of any blocks it doesn't recognize. This process continues until *A* discovers a leaf it doesn't recognize, after which it asks *B* for the value associated with that leaf. Unlike the basic form of anti-entropy, which required $O(n)$ communication for two nodes storing *n* values, this form requires only $O(d \log n)$ communication, where *d* is the number of differences between the values the two nodes are storing.

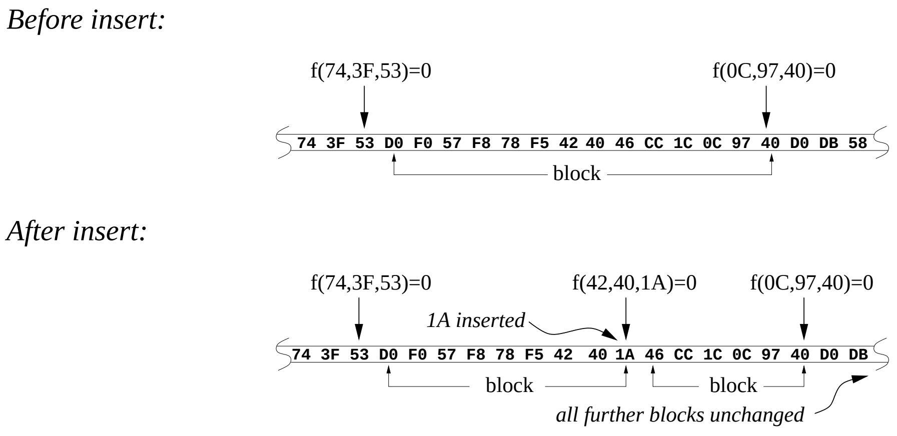

Before insert:
f(74,3F,53)=0
f(0C,97,40)=0
74 3F 53 D0 F0 57 F8 78 F5 42 40 46 CC 1C 0C 97 40 D0 DB 58
block
After insert:
f(74,3F,53)=0
f(42,40,1A)=0
f(0C,97,40)=0
1A inserted
74 3F 53 D0 F0 57 F8 78 F5 42 40 1A 46 CC 1C 0C 97 40 D0 DB
block
block
all further blocks unchanged

Figure 4.5: *Picking block boundaries using Rabin functions.* An insert may change the block in which it occurs, split an existing block into two (shown here), or cause two existing blocks to be combined; the remainder of the blocks in a stream remain the same.

synchronized nodes. However, *A* and *B* only realize this cost savings if all of their differences are at the end of the sorted set of $(k, H(v))$ pairs. A single difference, for example, at the head of this set will offset the block boundaries in the Merkle tree, causing all leaf and interior blocks to differ, and increase the cost back to $O(n)$. To fix this problem, we need to find a way to compute the blocks in the Merkle tree such that they do not change much after insertions.

Picking block boundaries with Rabin functions A Rabin function [Rab81], is a function f mapping n one-byte inputs uniformly and randomly to the set $\{0,\ldots,m-1\}$. In other words,

$$f:\overbrace{B\times B\times\cdots\times B}^{n\ \text{times}}\longrightarrow\{0,\ldots,m-1\}$$

where B is the set of possible byte values. We can place a block boundary before byte i in a file if the value of f on the n bytes proceeding byte i is 0. Since f is uniform and random, we expect to evaluate it on average m times before finding a zero; thus the expected block size from this technique is m. In addition, we set a minimum and maximum block size (such as $m/2$ and $2m$) to avoid choosing blocks that are too small or large. Figure 4.5 shows an example of computing block boundaries in this manner for $n = 3$. There, $f(0\text{C}, 97, 40) = 0$, so the block in the upper half of the figure ends on byte 40.

The benefit in picking block boundaries by the value of the underlying data is illustrated in the lower half of Figure 4.5, where the byte 1A has been inserted into the stream. In this example,

$f(42,40,1\mathrm{A}) = 0$, so this change introduces a new boundary, splitting the original block in two. However, since $n = 3$, the boundaries of all blocks three or more positions after the change are unaffected; in particular, the next block still starts with the byte value D0. In general, the insertion, modification, or deletion of any byte either changes only the block in which it occurs, splits that block into two, or combines that block with one of its neighbors. All other blocks in the stream are unaffected.

The application of Rabin functions to our anti-entropy problem is straightforward. Given our sorted set of $(k,H(v))$ pairs, we pick block boundaries in the set using Rabin functions. Next, we use Rabin functions again to compute the boundaries of the indirect blocks at the next level up and continue in this manner until we have only one indirect block at a level; this block is then the root of the tree.

Picking block boundaries by prefix Another technique for picking block boundaries is to group all of the $(k,H(v))$ pairs with a common prefix into a block. If this block is larger than a certain size, we lengthen the prefix that a pair must match by one bit, splitting the block into two. In this way, our Merkle tree becomes a trie. Tries also handle insertions and deletions well; note that when we add a new $(k,H(v))$ pair to a trie or remove an existing one, the only blocks that change are those that share prefixes with that pair.

Bamboo uses this method of picking block boundaries, as the code for it is simpler than that using Rabin functions. Furthermore, Bamboo uses 6-bit digits when computing prefixes for the following reasons. We can summarize each $(k,H(v))$ pair by its SHA-1 hash, and we can store 64 SHA-1 hashes in a single, 1,500-byte network packet with 220 bytes left over for headers and other information. As a result, we can use 6-bit digits in our prefixes (as $2^6 = 64$), giving the trie a branching factor of 64 and reducing its maximum height by a factor of 6, and still fit each interior node of the trie into a single network packet.

The choice of sorting order Above we mentioned that it is necessary to sort a node's $(k,H(v))$ pairs before building a Merkle tree over them. Otherwise, two nodes with the same values could produce different trees. Here we show that the sorting order we choose is also important.

For example, consider that pairs are sorted in the obvious way, lexigraphically by the bits of each $(k,H(v))$ pair, and consider two nodes A and B. Further consider that A temporarily loses connectivity, perhaps because it crashed, and during this period a number of additional tuples are put into the system. What happens when A regains connectivity and synchronizes its database with

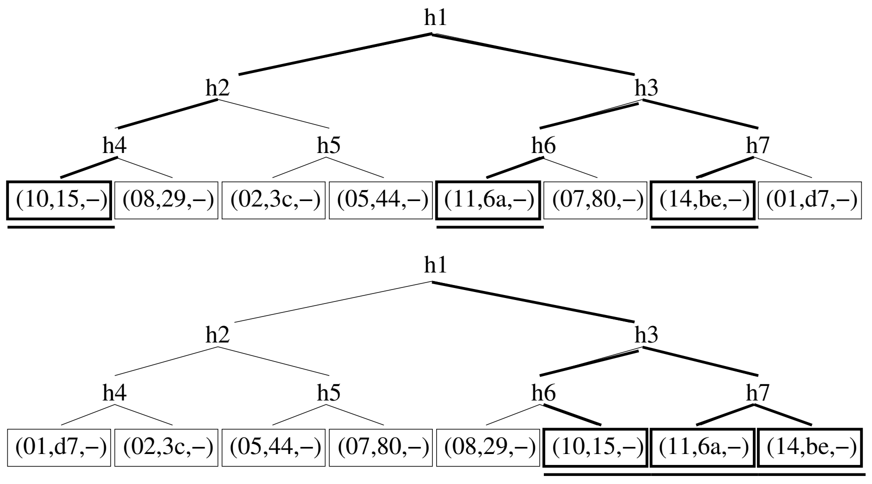


Figure 4.6: *The importance of sort order.* Above, the (timestamp, key, value) tuples have been sorted by their keys before building the Merkle tree, and the new blocks—shown underlined—are found through largely distinct paths from the root. Below, the tuples have been sorted by timestamp, and the new blocks share most of their paths.

B?

The upper half of Figure 4.6 illustrates this example. Since the keys used in a DHT are generally the output of a secure hash function, they are effectively chosen uniformly at random. They will thus fall in random places in the sorted order and thereby end up in random leaves of the Merkle tree. As such, for each mismatched key, several distinct intermediate hashes must be transmitted and compared, at a cost of several round trip times for each missing value corrected. Since temporarily losing connectivity is a common reason that nodes fall out of synchronization, this concern is an important one.

Assume instead that the DHT node at which a put originates timestamps the key and value put as they enter the system, and assume that we now sort $(t,k,H(v))$ tuples by their timestamps, t, breaking ties using the previous sort order. This change is illustrated in the lower half of Figure 4.6. Note that if A and B were synchronized before A lost connectivity, then the new values put into the system while A was unavailable will all end up in a continuous range of the leaves of the Merkle tree. Unlike when tuples were sorted by key, then, the differences in their Merkle trees should be confined to the side with the more recent tuples, and each traversal from the root to the leaves will lead to the discovery of many differences, resulting in far fewer round-trip times than before.

Of course, poor clock synchronization can reduce the benefits of this technique, but it is almost inconceivable that it would perform worse than sorting by key—the clocks of the nodes in the DHT would have to be set to random values.

One challenge that does arise in sorting by timestamp is that there must now be a separate Merkle tree for each shared subrange of the keyspace. When sorting by key, each node can build a single Merkle tree; the differences in the ranges covered by these trees will be discovered during the process of synchronization. In contrast, when sorting by timestamp, each node builds $r-1$ trees, one for each unique subrange of keys that it shares with its neighbors.

One more slight concern remains: when violations of transitive connectivity of the kind discussed in Chapter 6 occur, nodes will disagree with each other as to which other nodes should be in their leaf sets. In such cases, they will also disagree as to the subranges for which Merkle trees should be built. Fortunately, any two nodes that can connect to the same subset of their proper neighbors in the DHT will agree about these subranges and will still be able to synchronize. The others must wait until the anomaly heals. In practice, such anomalies do not persist forever, and we have not found this limitation to be a problem on our PlanetLab deployment.

4.3.2 Discarding Unwanted Values

As we noted above, as long as one replica in a set contains a value, they all will eventually. We must also protect against the case where no replica contains a value, however. This circumstance can occur for a number of reasons. First, a partition in the DHT may cause a value to be replicated on the wrong set of hosts; when the partition heals,[2] we would like the value to be moved to the correct set of hosts. Consider Figure 4.7, for example. Normally, the three UC Berkeley nodes are spread around the ring, and none of them are in $R(k)$; when UC Berkeley's connection to the Internet fails, however, they form a three-node ring of their own, and all three are in $R(k)$. Any puts on key k that occur during this period from within Berkeley will thus be stored only on the Berkeley nodes. When the partition heals, however, they must be moved to the original $R(k)$, even though none of the Berkeley nodes are in that set.

The problem of puts that occur during partitions is actually a special case of the more general problem of massive join events. Consider again Figure 4.7, and assume that originally the DHT only contained the three Berkeley nodes. Now consider that all of the grey nodes join simultaneously. Just as before, some values stored on the Berkeley nodes must be moved to replica sets of which they are not members, motivating the need for an additional mechanism beyond replica

[2]A Bamboo node periodically checks for and corrects the effects of network partitions by trying to rejoin the network through nodes that were its neighbors in the past but to which it has since lost connectivity.

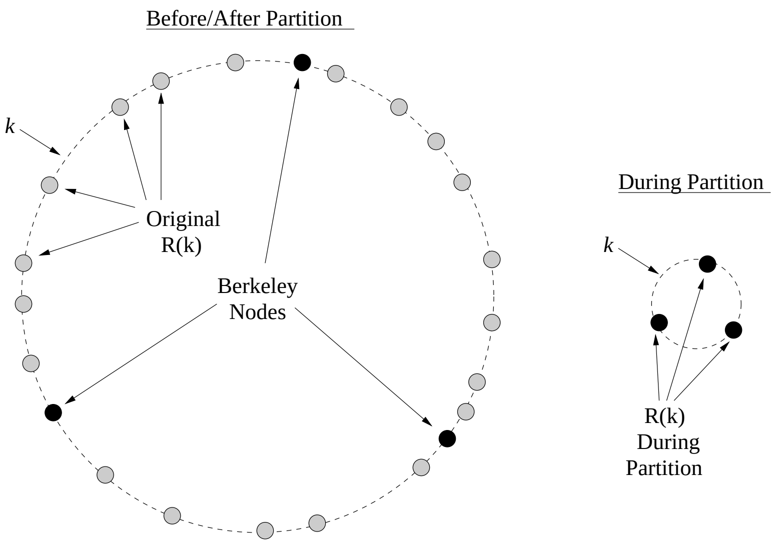


Figure 4.7: *The need for the discard algorithm.* The figure on the left shows a Bamboo ring; the three black nodes are all at UC Berkeley. When Berkeley's connection to the Internet fails, these three nodes can no longer communicate with any others, so they assume the others have failed and form a three-node ring of their own. Any puts on key k during this partition are stored on the Berkeley nodes. Once the partition heals, however, these values must be moved to the original $R(k)$ shown on the left. Replica synchronization will not accomplish this movement, however, as it only involves communication within $R(k)$, and this set no longer includes any of the Berkeley nodes.

synchronization.[3]

To fix this problem, we introduce the the discard algorithm, which allows nodes not in $R(k)$ to safely discard values stored under k. Our original implementation of the discard algorithm worked as follows: when a node that was no longer in $R(k)$ noticed that it was storing a key-value pair (k,v), it simply re-put it. While this approach is very easy to implement, it is not terribly efficient; in the case of massive joins described above, each original replica pushes each value to each new one, leading to a $O(r^2)$ cost.

A simple optimization for the discard algorithm is as follows: rather than re-put a value it should no longer be storing, a node sends the value to a random node in the current replica set. The current Bamboo implementation uses this procedure. We note that this procedure is efficient, sending only one message per original replica. It is also reasonably robust: consider a massive join event, in which all of the original replica set is replaced by new nodes. Table 4.1 shows the probability that at least m new replicas receive a value if n old replicas perform the discard algorithm

[3]Our discovery of this need was actually due to massive join events as described here. Such events were one part of a larger testing framework we used to debug the early storage management code.

n \ m	1	2	3	4	5	6	7	8
1	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	1.000	0.875	0.000	0.000	0.000	0.000	0.000	0.000
3	1.000	0.984	0.656	0.000	0.000	0.000	0.000	0.000
4	1.000	0.998	0.902	0.410	0.000	0.000	0.000	0.000
5	1.000	1.000	0.974	0.718	0.205	0.000	0.000	0.000
6	1.000	1.000	0.993	0.878	0.461	0.077	0.000	0.000
7	1.000	1.000	0.998	0.950	0.670	0.221	0.020	0.000
8	1.000	1.000	1.000	0.980	0.810	0.389	0.070	0.002

Table 4.1: *The effectiveness of the discard algorithm.* After a massive join event, where the entire original replica set is replaced by new nodes, each original replica sends each value for which it is no longer responsible to a random new replica. This table shows the probability that at least *m* new replicas receive a value if *n* old replicas perform the discard algorithm. For example, if five old replicas do so, there is a 71.8% chance that at least four distinct new replicas will receive a value. (The remaining replicas will eventually get one through the replica synchronization process.)

for a system with $r = 8$. For example, if five old replicas perform the discard algorithm, there is a 71.8% chance that at least four distinct new replicas will receive a value. (The remaining replicas will eventually get one through the replica synchronization process.)

4.4 Handling Mutable Data

DHTs generally offer only eventually-consistent semantics: if the system is stable, eventually all clients will see all values put. Moreover, using insights from the epidemic literature, it is reasonably easy to support some degree of mutable data.

A straightforward approach to making values in the DHT mutable would be that each new value put overrides any previous value under the same key. While this approach at first sounds reasonable, it is difficult to implement in the face of concurrent puts to the same key issued from different nodes.

In special circumstances, however, the problem is tractable: if there exists some total ordering over all possible values, DHT nodes can use this ordering to choose between them, and so long as this ordering is a deterministic function of the values themselves, all nodes in the DHT will eventually agree as to the current value under each key.

As an example of this approach, a remove operation is implemented in Bamboo in the following manner. Each value put into the system may contain the SHA-1 hash of a unique secret,

s. Moreover, we allow a certain class of values called *removes*. If a DHT node has a put $(k, v, H(s))$, and it learns about a remove $(k, H(v), s)$, it always discards the put in favor of the remove. In this way, a value can be removed from the system by simply putting a corresponding remove; in the total ordering used, removes always dominate their corresponding puts.

Note that if the system did not store removes, removed values could be resurrected as follows. Consider a node A that crashes, and consider that while A is down a value it is storing is removed from all remaining replicas. When A comes back up, the replica synchronization and discard protocols will propagate the value back onto A's neighbors, effectively re-inserting it into the system. By storing the remove like any other value, and by always preferring a remove over its associated put, we prevent such resurrections.

We discovered the idea of removes from the epidemic literature, where they are called *death certificates* [DGH+87]. That literature also contains a caution: a DHT node cannot discard any remove for which a put may still remain in the system. Since puts may exist on crashed nodes that cannot be contacted, discarding a remove must thus be done with care. In Bamboo, all values put into the system contain a time-to-live (TTL), and a node discards a value when this time passes. To safely remove a value, then, one must simply assure that the TTL for a remove is longer than the TTL remaining for its corresponding put.

4.5 Related Work

We developed the Bamboo replica synchronization and discard algorithms concurrently with, but independently from, Cates [Cat03], who was working on the DHash system at the time. Our replica synchronization algorithm is virtually identical to his local maintenance algorithm, and our discard algorithm is virtually identical to his global maintenance algorithm. Since Cates was working on Chord, $R(k)$ corresponds to the r successors of k, rather than the $r/2$ predecessors and successors of it. Also, DHash uses erasure codes instead of replication for redundancy. For the purposes of this chapter, these differences are largely irrelevant, however.

Our use of Merkle trees in synchronization followed from our use of them in Pond [REG+03], the OceanStore [KBC+00] prototype. Pond is a versioned file system, and it uses Merkle trees to certify the contents of each version and to prevent storing data shared between multiple versions multiple times. These same Merkle trees can be used to detect differences between versions, and that feature lead to our insight that they could be used for synchronization in Bamboo.

At the time, we had also just finished our work on Value-Based Web Caching [RLB03], where we used Rabin functions to chose block boundaries for efficient duplicate transfer elimination in HTTP. (Rabin functions have also been used elsewhere for similar purposes, including in file systems [MCM01] and packet-level network compression [SW00].) We developed the first replica synchronization algorithm in Bamboo by combining Merkle trees and Rabin functions as described in this chapter.

Messages from the OceanStore developers' mailing list [Rhe03a, Rhe03b] show that we had the basics of replica synchronization and the discard algorithm working in the Bamboo code base on April 18, 2003. Shortly thereafter, in May 2005, Cates' thesis was published, and we realized he had come up with a very similar solution to roughly the same problem. After reading his thesis, we switched from using Rabin functions to using tries to pick block boundaries, but we made no other changes to our algorithms.

A remaining difference between DHash and Bamboo is that DHash sorts values by key rather than timestamp. As discussed above, we believe this is less efficient, but only intuitively so; we have not compared the two in simulation, for example.

4.6 Future Work

Above we have discussed the existing Bamboo storage layer. In this section we present several improvements we plan to make in the future.

4.6.1 A Better Discard Algorithm

We have shown through analysis that the existing discard algorithm in Bamboo works well when none of the original replicas for a value remain in the replica set after a massive join event. However, in that analysis we assumed that all original replicas survive long enough to discard each value for which they are no longer responsible. In the case that the DHT is storing many values, however, some replicas may leave the system before transferring all of their state.

Consider that only one original replica survives long enough to perform the discard algorithm for a value. In the existing algorithm, only a single new replica will receive the value, and the system will be vulnerable to a single failure until the replica synchronization algorithm copies the value across the replica set.

A better algorithm for discarding values is clear from the epidemic literature: if we were to use rumor-mongering with $p = 1$ for discarding values, each original node would continue to send a value to new replicas until finding one that already has the value (and each node that receives a value would do the same). According to the epidemic literature, this change would increase the fraction of new replicas that receive a value in this example to 80% on expectation, for only a traffic cost of $1.74r$. Moreover, in the case where all of the original replicas survive, the traffic cost would still only increase over the existing algorithm by this constant factor. We thus see this change as a promising one for Bamboo (as well as DHash).

4.6.2 Accounting for Spatial Distributions

In our analysis above, we have ignored the fact that some nodes in a replica set are closer to each other in network latency than others. Since the replica synchronization algorithm requires multiple round trips, nearby replicas should be able to synchronize more quickly than remote ones. Also, synchronizing nearby replicas uses less total network resources. In general, then, we would prefer to synchronize nearby members of each replica set more often. There are limits to which this general goal can be achieved and still produce timely convergence, but the results in the epidemic literature are nonetheless surprising.

Consider a system where all the hosts lie along a linear network. If each host talks only to its immediate neighbors, the number of cycles to converge using anti-entropy is $O(N)$, and the average convergence traffic per link per cycle is $O(1)$. On the other hand, if each host chooses its anti-entropy partners randomly, convergence happens in $O(\log N)$ cycles and the traffic per link per cycle is $O(N)$. These two strategies represent extreme choices in replica selection.

Kempe, Kleinberg, and Demers showed that there is also a middle ground; for hosts arranged with uniform density in Euclidean space, they proved that using a special selection criteria a new tuple introduced at one host will reach all hosts within distance d of its origin in $O(\log^{1+\varepsilon} d)$ time [KKD01]. In practice, the effects of such spatial bias can be dramatic: using a topology modeled on the Xerox corporate intranet, Demers et al. showed that by choosing anti-entropy partners according to distance, convergence time increased by less than a factor of two while reducing traffic across a bottleneck trans-Atlantic link by over a factor of 30.

Unlike with routing table neighbors, the members of a node's leaf set cannot be chosen for proximity; they are instead fixed by a node's position in the ring. As such, we view spatially-aware epidemics as a promising technique for reducing the load imposed by consistency traffic in DHTs.

4.6.3 Further Reducing the Cost of Temporary Failures

As discussed in Chapter 2, every time a node permanently leaves the DHT, we must copy data to other nodes in order to restore replication. Furthermore, when new nodes join, existing nodes will no longer be responsible for some of the data they are storing, and after discarding that data to new nodes, they can delete it from their local storage.

A problem with current algorithms, however, is that they do not distinguish permanent failures from temporary ones. If a node goes offline temporarily, the DHT will begin re-replicating all of the data it stored immediately. If this node then recovers, this replication traffic will have merely wasted resources. A related problem is that of a short join: when a new node joins the DHT, replicas are moved onto it and deleted from existing nodes; if the new node departs shortly thereafter, replicas will have to be recreated on the same nodes that just deleted them.

Although it is not possible to precisely distinguish temporary failures from permanent ones, we can approximate the distinction by introducing a *repair threshold* [BTC$^+$04]: by setting an original replication factor of n, but only re-replicating data when the available redundancy falls below some $m < n$, we can prevent many temporary failures from triggering repair.

The DHash group has recently developed an interesting adaptation of this idea to storage management algorithms like those used in DHash and Bamboo [DS]. They start by assuming the system has a roughly constant size over time, and by assuming each node is part of the system some fraction a of the time. They then assume that no node ever removes a replica from its local storage, and they modify the replica synchronization algorithm to only create a new replica for a value if less than r are available in the $2r$ closest nodes in the ring. Using this technique, they show that the expected degree of replication per value converges to $2r/a$, a reasonable cost for a system such as PlanetLab where each node has relatively high availability.

In such a system, a temporary failure of a single node causes no data movement unless it causes the number of replicas for an item to fall below r. Likewise, when a new node joins, no data is moved onto it unless its join pushes enough replicas out of the $2r$ closest nodes to some key to cause the replication factor to fall below r. In both cases, it is unlikely that data will be moved, reducing much of the cost incurred by the current design due to temporary failures and joins.

One complication of this design is that the replica synchronization algorithm must be modified to keep track of *which* r or more nodes are storing each value. In DHash, this is implemented by storing that information locally with the values. Furthermore, if the replication factor for an item is greater than r, no new replicas should be created. To prevent their creation, nodes that have replicas of some value must hide this fact from those that do not during replica synchronization. (Otherwise, the latter will pull a copy from the former upon discovering they are missing each such value.) In DHash, this hiding is accomplished by building a separate Merkle tree for each neighbor, increasing the memory cost of storing the Merkle trees by a factor of $2r$. Because this cost is too high, DHash now builds Merkle trees on demand, reducing how often synchronization can be performed, and only synchronizes with a few neighbors at a time.

Despite the additional difficulties involved in implementing this extension to replica synchronization, we are still in favor of it, as it saves a great deal of bandwidth, which is usually the limiting resource in a DHT.

4.6.4 Controlled Performance Studies

One limitation of this chapter is that we have yet to perform a careful study of the effectiveness of the techniques described here. There are several reasons we have not done so. First, Cates performed a study of the DHash algorithms in his work, and our algorithms are quite similar to those, so a performance study seems at least a little redundant. That said, it would be nice to compare the relative benefits of sorting values by timestamp rather than key. Second, our algorithms seem to work in practice. As we will show in a Chapter 5, our OpenDHT deployment on 200–300 PlanetLab seems to store data effectively despite the churn it experiences. Finally, a performance evaluation has simply been low on our list of priorities. The latency of get operations on OpenDHT, for example, matters much more to our users at the moment (see Chapter 7). As the system becomes more popular and stores more and more data, however, the efficiency of the storage system will become important. At that point we hope to profile the system as it stands, and we also plan to implement and test the performance optimizations listed above.

4.7 Summary and Discussion

In this chapter we have shown a successful combination of DHTs and epidemic algorithms. We believe the two are a natural match. Epidemic algorithms provide a robust, efficient, and simple way to maintain consistency between replicas, but do not naturally support a system in which each data value is replicated by only a subset of nodes. In contrast, DHTs provide a partitioning suitable for choosing a set of replicas for each data value in a system, but require an algorithm for maintaining their consistency.

We are not the first to recognize the value of partitioning a large system to limit the degree of replication; Demers et al. noted it in their work. We believe we were the first to note the utility of a DHT for this purpose, however. As far as we can tell, while Cates clearly used epidemic techniques, he appears to have been unaware of the epidemic literature and was apparently unaware of their status as such. Moreover, while the Kelips [GBL+03] system uses epidemic techniques to build a DHT lookup layer, they do not discuss the application of these techniques to replica management.

Chapter 5

OpenDHT

In Chapters 3 and 4, we presented the design and implementation of the lookup and storage layers of a DHT. By going to the Bamboo web page and downloading these components, would-be application developers obtain a useful building block on which to construct their systems. Nonetheless, maintaining a running DHT remains non-trivial. At least with current technology, DHT nodes cannot run behind NATs, so a group of machines on the public Internet on which to run the DHT must be acquired. Scripts must be written to keep the DHT code running on these machines, and someone must be around to reboot them in case they experience kernel bugs or fix them when they experience hardware failures.

As DHT-based applications proliferate, it is thus natural to ask whether every application needs its own DHT deployment, or whether a shared deployment could amortize this operational effort across many different applications. While some applications do in fact make extremely sophisticated use of DHTs, many more access them through such a narrow interface that it is reasonable to expect they might benefit from a shared infrastructure.

In this chapter, we report on our efforts to design and build OpenDHT (formerly named OpenHash [KRRS04]), a shared DHT deployment. Specifically, our goal is to provide a free, public DHT service that runs on PlanetLab [B$^+$04] today. Longer-term, as we consider later in this chapter, we envision that this free service could evolve into a competitive commercial market in DHT service.

Figure 5.1 shows the high-level architecture of OpenDHT. Infrastructure nodes run the OpenDHT server code. Clients are nodes *outside* the set of infrastructure nodes; they run application code that invokes the OpenDHT service using RPC. Besides participating in the DHT's routing and storage, each OpenDHT node also acts as a *gateway* through which it accepts RPCs from clients.

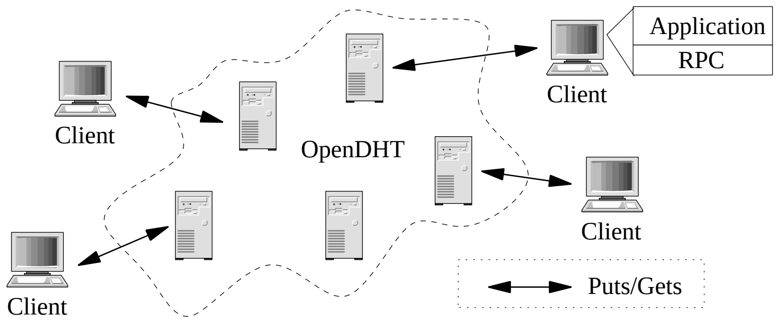


Figure 5.1: OpenDHT Architecture.

Because OpenDHT operates on a set of infrastructure nodes, no application need concern itself with DHT deployment, but neither can it run application-specific code on these infrastructure nodes. This is quite different than most other uses of DHTs, in which the DHT code is invoked as a library on each of the nodes running the application. The library approach is very flexible, as one can put application-specific functionality on each of the DHT nodes, but each application must deploy its own DHT. The service approach adopted by OpenDHT offers the opposite tradeoff: less flexibility in return for less deployment burden. OpenDHT provides a home for applications more suited to this compromise.

The service approach not only offers a different tradeoff; it also poses different design challenges. Because of its shared nature, building OpenDHT is not the same as merely deploying an existing DHT implementation on PlanetLab. OpenDHT is shared in two different senses: there is sharing both among applications and among clients, and each raises a new design problem.

First, for OpenDHT to be shared effectively by many different applications, its interface must balance the conflicting goals of generality and ease-of-use. Generality is necessary to meet the needs of a broad spectrum of applications, but the interface should also be easy for simple clients to use. Ease-of-use argues for a fairly simple primitive, while generality (in the extreme) suggests giving raw access to the operating system (as is done in PlanetLab).[1] It is hard to quantify both ease-of-use and generality, so we rely on our early experience with OpenDHT applications to evaluate our design decisions. Not knowing what applications are likely to emerge, we can only conjecture about the required degree of generality.

Second, for OpenDHT to be shared by many mutually untrusting clients without their

[1]One might argue that PlanetLab solves the problems we are posing by providing extreme resource control and a general interface. But PlanetLab is hard for simple clients to use, in that every application must install software on each host and ensure its continued operation. For many of the simple applications we describe in Section 5.4.3, this effort would be inappropriately burdensome.

unduly interfering with each other, system resources must be allocated with care. While ample prior work has investigated bandwidth and CPU allocation in shared settings, storage allocation has been studied less thoroughly. In particular, there is a delicate tradeoff between fairness and flexibility: the system shouldn't unnecessarily restrict the behavior of clients by imposing arbitrary and strict quotas, but it should also ensure that all clients have access to their fair share of service. Here we can evaluate prospective designs more quantitatively, and we do so with extensive simulations.

We summarize our solutions to these two design problems in Section 5.1. We then address in significantly more detail the OpenDHT interface (Section 5.2) and storage allocation algorithm (Section 5.3). Section 5.4 describes our early deployment experience, both in terms of raw performance and availability numbers, and the variety of applications currently using the system. Section 5.5 concludes with a discussion of various economic concerns that may affect the design and deployment of services like OpenDHT.

5.1 Overview of Design

Before delving into the details of OpenDHT in subsequent sections, we first describe the fundamental rationale for the designs we chose for the system's interface and storage allocation mechanism.

5.1.1 Interface

In designing OpenDHT, we have the conflicting goals of generality and ease-of-use (which we also refer to as simplicity). There are three broad classes of interfaces in the DHT literature, and they each occupy very different places on the generality/simplicity spectrum (a slightly different taxonomy is described in [DZD$^+$03]). Given a key, these interfaces provide three very different capabilities:

routing Provides general access to the DHT node responsible for the input key, and to each node along the DHT routing path.

lookup Provides general access to the DHT node responsible for the input key.

storage Directly supports the put(key, value) and get(key) operations by routing them to the DHT node responsible for the input key, but exposes no other interface.

The *routing* model is the most general interface of the three; a client is allowed to invoke arbitrary code at the endpoint and at every node along the DHT path towards that endpoint (either through upcalls or iterative routing). This interface has been useful in implementing DHT-based multicast [CDK+03a] and anycast [ZHS+04].

The *lookup* model is somewhat less general, only allowing code invocation on the endpoint. This has been used for query processing [HHL+03], file systems [DKK+01, MMGC02], and packet forwarding [SAZ+02].

The true power of the routing and lookup interfaces lies in the application-specific code running on the DHT nodes. While the DHT provides routing to the appropriate nodes, it is the application-specific code that does the real work, either at each hop en route (routing) or only at the destination (lookup). For example, such code can handle forwarding of packets (e.g., multicast and *i*3 [SAZ+02]) or data processing (e.g., query processing).

The *storage* model is by far the least flexible, allowing no access to application-specific code and only providing the put/get primitives. This lack of flexibility greatly limits the spectrum of applications it can support, but in return this interface has two advantages: it is simple for the service to support, in that the DHT infrastructure need not deal with the vagaries of application-specific code running on each of its nodes, and it is also simple for application developers and deployers to use, freeing them from the burden of operating a DHT when all they want is a simple put/get interface.

In the design of OpenDHT, we place a high premium on simplicity. We want an infrastructure that is simple to operate, and a service that simple clients can use. Thus the storage model, with its simple put/get interface, seems most appropriate. To get around its limited functionality, we use a novel client library, Recursive Distributed Rendezvous (ReDiR), which we describe in detail in Section 5.2.2. ReDiR, in conjunction with OpenDHT, provides the equivalent of a lookup interface for any arbitrary set of machines (inside or outside OpenDHT itself). Thus clients using ReDiR achieve the flexibility of the lookup interface, albeit with a small loss of efficiency (which we describe later).

Our design choice reflects our priorities, but one can certainly imagine other choices. For instance, one could run a shared DHT on PlanetLab, with the DHT providing the routing service and PlanetLab allowing developers to run application-specific code on individual nodes. This would relieve these developers of operating the DHT, and still provide them with all the flexibility of the routing interface, but require careful management of the application-specific code introduced on the various PlanetLab nodes. We hope others explore this portion of the design space, but we are primarily interested in facilitating simple clients with a simple infrastructure, and so we chose a different design.

While there are no cut-and-dried metrics for simplicity and generality, early evidence suggests we have navigated the tradeoff between the two well. As we describe in greater detail in Section 5.4.1, OpenDHT is highly robust, and we firmly believe that the relative simplicity of the system has been essential to achieving such robustness. While generality is similarly difficult to assess, in Table 5.4 we offer a catalog of the diverse applications built on OpenDHT as evidence of the system's broad utility.

5.1.2 Storage Allocation

OpenDHT is essentially a public storage facility. As observed in [RH03, BMP03], if such a system offers the persistent storage semantics typical of traditional file systems, the system will eventually fill up with orphaned data. Garbage collection of this unwanted data seems difficult to do efficiently. To frame the discussion, we consider the solution to this problem proposed as part of the Palimpsest shared public storage system [RH03]. Palimpsest uses a novel revolving-door technique in which, when the disk is full, new stores push out the old. To keep their data in the system, clients re-put frequently enough so that it is never flushed; the required re-put rate depends on the total offered load on that storage node. Palimpsest uses per-put charging, which in this model becomes an elegantly simple form of congestion pricing to provide fairness between users (those willing to pay more get more).

While we agree with the basic premise that public storage facilities should not provide unboundedly persistent storage, we are reluctant to require clients to monitor the current offered load in order to know how often to re-put their data. This adaptive monitoring is complicated and requires that clients run continuously. In addition, Palimpsest relies on charging to enforce some degree of fairness; since OpenDHT is currently deployed in an environment where such charging is both impractical and impolitic, we wanted a way to achieve fairness without an explicit economic incentive.

Our goals for the OpenDHT storage allocation algorithm are as follows. First, to simplify life for its clients, OpenDHT should offer storage with a definite time-to-live (TTL). A client should know exactly when it must re-store its puts in order to keep them stored, so rather than adapting (as in Palimpsest), the client can merely set simple timers or forget its data altogether (if, for instance, the application's need for the data will expire before the data itself).

Second, the allocation of storage across clients should be “fair” without invoking explicit charging. By fair we mean that, upon overload, each client has “equal” access to storage.[2] Moreover, we also mean fair in the work-conserving sense; OpenDHT should allow for full utilization of the storage available (thereby precluding quota-like policies), and should restrict clients *only* when it is overloaded.

Finally, OpenDHT should prevent *starvation* by ensuring a minimal rate at which puts can be accepted at all times. Without such a requirement, the system could allocate all its storage (fairly) for an arbitrarily long TTL, and then reject all storage requests for the duration of that TTL. Such “bursty” availability of storage would present an undue burden on OpenDHT clients.

In Section 5.3 we present an algorithm that meets the above goals.

The preceding was an overview of our design. We next consider the details of the OpenDHT client interface, and thereafter, the details of storage allocation in OpenDHT.

5.2 Interface

One challenge to providing a shared DHT infrastructure is designing an interface that satisfies the needs of a sufficient variety of applications to justify the shared deployment. OpenDHT addresses this challenge two ways. First, a put/get interface makes writing simple applications easy yet still supports a broad range of storage applications. Second, the use of a client-side library called ReDiR allows more sophisticated interfaces to be built atop the base put/get interface. In this section we discuss the design of these interfaces. Section 5.4 presents their performance and use.

5.2.1 The put/get API

The OpenDHT put/get interface supports a range of application needs, from storage in the style of the Cooperative File System (CFS) [DKK$^+$01] to naming and rendezvous in the style of the Host Identity Protocol (HIP) [MNJH04] and instant messaging.

The design goals behind the put/get interface are as follows. First, simple OpenDHT applications should be simple to write. The value of a shared DHT rests in large part on how easy it is to use. OpenDHT can be accessed using either Sun RPC over TCP or XML RPC over HTTP; as such it easy to use from most programming languages and works from behind most firewalls and NATs. A Python program that reads a key and value from the console and puts them into the DHT

[2]As in fair queuing, we can of course impose weighted fairness, where some clients receive a larger share of storage than others, for policy or contractual reasons. We do not pursue this idea here, but it would require only minor changes to our allocation mechanism.

is only nine lines long; the complementary get program is only eleven.

Second, OpenDHT should not restrict key choice. Previous schemes for authentication of values stored in a DHT require a particular relationship between the value and the key under which it is stored (e.g., [DKK+01, DR01]). Already we know of applications that have key choice requirements that are incompatible with such restrictions; the prefix hash tree (PHT) [RRHS04, CRR+05] is one example. It would be unwise to impose similar restrictions on future applications.

Third, OpenDHT should provide authentication for clients that need it. A client may wish to verify that an authorized entity wrote a value under a particular key or to protect its own values from overwriting by other clients. As we describe below, certain attacks cannot be prevented without support for authentication in the DHT. Of course, our simplicity goal demands that authentication be only an option, not a requirement.

The current OpenDHT deployment meets the first two of these design goals (simplicity and key choice) and has some support for the third (authentication). In what follows, we describe the current interface in detail, then describe two planned interfaces that better support authentication. Table 5.1 summarizes all three interfaces. Throughout, we refer to OpenDHT keys by k; these are 160-bit values, often the output of the SHA-1 hash function (denoted by H), though applications may assign keys in whatever fashion they choose. Values, denoted v, are variable-length, up to a maximum of 1 kB in size. All values are stored for a bounded time period only; a client specifies this period either as a TTL or an expiration time, depending on the interface.

Finally, we note that under all three interfaces, OpenDHT provides only eventual consistency. In the case of network partitions or excessive churn, the system may fail to return values that have been put or continue to return values that have been removed. Imperfect clock synchronization in the DHT may also cause values to expire at some replicas before others, leaving small windows where replicas return different results. While such temporary inconsistencies in theory limit the set of applications that can be built on OpenDHT, they have not been a problem to date.

The Current Interface

A put in OpenDHT is uniquely identified by the triple of a key, a value, and the SHA-1 hash of a client-chosen random secret up to 40 bytes in length. If multiple puts have the same key and/or value, all are stored by the DHT. A put with the same key, value, and secret hash as an existing put refreshes its TTL. A get takes a key and returns all values stored under that key, along with their associated secret hashes and remaining TTLs. An iterator interface is provided in case

Procedure	Functionality
$\mathit{put}(k,v,H(s),t)$	Write (k,v) for TTL t; can be removed with secret s
$\mathit{get}(k)$ returns $\{(v,H(s),t)\}$	Read all v stored under k; returned value(s) unauthenticated
$\mathit{remove}(k,H(v),s,t)$	Remove (k,v) put with secret s; $t >$ than TTL remaining for put
$\mathit{put\text{-}immut}(k,v,t)$	Write (k,v) for TTL t; immutable $(k = H(v))$
$\mathit{get\text{-}immut}(k)$ returns (v,t)	Read v stored under k; returned value immutable
$\mathit{put\text{-}auth}(k,v,n,t,K_P,\sigma)$	Write (k,v), expires at t; public key K_P; private key K_S; can be removed using nonce n; $\sigma = \{H(k,v,n,t)\}_{K_S}$
$\mathit{get\text{-}auth}(k,H(K_P))$ returns $\{(v,n,t,\sigma)\}$	Read v stored under $(k,H(K_P))$; returned value authenticated
$\mathit{remove\text{-}auth}(k,H(v),n,t,K_P,\sigma)$	Remove (k,v) with nonce n; parameters as for *put-auth*

Table 5.1: The put/get interface. $H(x)$ is the SHA-1 hash of x.

there are many such values.

To remove a value, a client reveals the secret whose hash was provided in the put. A put with an empty secret hash cannot be removed. As discussed in Chapter 4, OpenDHT stores removes like puts, but a DHT node discards a put $(k, v, H(s))$ for which it has a corresponding remove. To prevent the DHT's replication algorithms from recovering this put when the remove's TTL expires, clients must ensure that the TTL on a remove is longer than the TTL remaining on the corresponding put. Once revealed in a remove, a secret should not be reused in subsequent puts. To allow other clients to remove a put, a client may include the encrypted secret as part of the put's value.

To change a value in the DHT, a client simply removes the old value and puts a new one. In the case where multiple clients perform this operation concurrently, several new values may end up stored in the DHT. In such cases, any client may apply an application-specific conflict resolution procedure to decide which of the new values to remove. So long as this procedure is a total ordering of the possible input values, it does not matter which client performs the removes (or even if they all do); the DHT will store the same value in the end in all cases. This approach is similar to that used by Bayou [PST$^+$97] to achieve eventual consistency.

Since OpenDHT stores all values put under a single key, puts are robust against *squatting*, in that there is no race to put first under a valuable key (e.g., H(“coca-cola.com”)). To allow others to authenticate their puts, clients may digitally sign the values they put into the DHT. In the current OpenDHT interface, however, such values remain vulnerable to a denial-of-service attack we term *drowning*: a malicious client may put a vast number of values under a key, all of which will be stored, and thereby force other clients to retrieve a vast number of such chaff values in the process of retrieving legitimate ones.

Planned Interfaces

Although the current put/get interface suffices for the applications built on OpenDHT today, we expect that as the system gains popularity developers will value protection against the drowning attack. Since this attack relies on forcing legitimate clients to sort through chaff values put into the DHT by malicious ones, it can only be thwarted if the DHT can recognize and reject such chaff. The two interfaces below present two different ways for the DHT to perform such access control.

Immutable puts: One authenticated interface we plan to add to OpenDHT is the immutable put/get interface used in CFS [DKK$^+$01] and Pond [REG$^+$03], for which the DHT only allows puts where $k = H(v)$. Clearly, such puts are robust against squatting and drowning. Immutable puts will not be removable; they will only expire. The main limitation of this model is that it restricts an application's ability to choose keys.

Signed puts: The second authenticated interface we plan to add to OpenDHT is one where values put are certified by a particular public key, as used for root blocks in CFS. In these puts, a client employs a public/private key pair, denoted K_P and K_S, respectively. We call $H(K_P)$ the *authenticator*.

In addition to a key and value, each put includes: a nonce n that can be used to remove the value later; an expiration time t in seconds since the epoch; K_P itself; and $\sigma = \{H(k,v,n,t)\}_{K_S}$, where $\{X\}_{K_S}$ denotes the digital signing of X with K_S. OpenDHT checks that the digital signature verifies using K_P; if not, the put is rejected. This invariant ensures that the client that sent a put knows K_S.

A get for an authenticated put specifies *both* k and $H(K_P)$, and returns only those values stored that match both k and $H(K_P)$. In other words, OpenDHT only returns values signed by the private key matching the public key whose hash is in the get request. Clients may thus protect themselves against the drowning attack by telling the DHT to return only values signed by an entity they trust.

To remove an authenticated put with (k,v,n), a client issues a remove request with $(k,H(v),n)$. As with the current interface, clients must take care that a remove expires after the corresponding put. To re-put a removed value, a client may use a new nonce $n' \neq n$.

We use expiration times rather than TTLs to prevent expired puts from being replayed by malicious clients. As with the current interface, puts with the same key and authenticator but different values will all be stored by the DHT, and a new put with the same key, authenticator, value, and nonce as an existing put refreshes its TTL.

Authenticated puts in OpenDHT are similar to those used for public-key blocks in CFS [DKK+01], for *sfrtags* in SFR [WBS04], for *fileIds* in PAST [DR01], and for AGUIDs in Pond [REG+03]. Like SFR and PAST, OpenDHT allows multiple data items to be stored using the same public key. Unlike CFS, SFR, and PAST, OpenDHT gives applications total freedom over key choice (a particular requirement in a generic DHT service).

5.2.2 ReDiR

While the put/get interface is simple and useful, it cannot meet the needs of all applications. Another popular DHT interface is *lookup*, which is summarized in Table 5.2. In this interface,

Procedure	Functionality
join(*host*, *id*, *namespace*)	adds (*host*, *id*) to the list of hosts providing functionality of *namespace*
lookup(*key*, *namespace*)	returns (*host*, *id*) in *namespace* whose *id* most immediately follows *key*

Table 5.2: The lookup interface provided using ReDiR.

nodes that wish to provide some service—packet forwarding, for example—*join* a DHT dedicated to that service. In joining, each node is associated with an identifier *id* chosen from a *key space*, generally $[0:2^{160})$. To find a service node, a client performs a lookup, which takes a key chosen from the identifier space and returns the node whose identifier most immediately follows the key; lookup is thus said to implement the successor relation.

For example, in *i*3 [SAZ$^+$02], service nodes provide a packet forwarding functionality to clients. Clients create (key, destination) pairs called triggers, where the destination is either another key or an IP address and port. A trigger (k,d) is stored on the service node returned by *lookup* (*k*), and this service node forwards all packets it receives for key *k* to d. Assuming, for example, that the nodes A through F in Figure 5.2 are *i*3 forwarding nodes, a trigger with key $B \leq k < C$ would be managed by service node C.

The difficulty with lookup for a DHT service is the functionality implemented by those nodes returned by the lookup function. Rather than install application-specific functionality into the service, thereby certainly increasing its complexity and possibly reducing its robustness, we prefer that such functionality be supported outside the DHT, while leveraging the DHT itself to perform lookups. OpenDHT accomplishes this separation through the use of a client-side library called ReDiR. (An alternative approach, where application-specific code may only be placed on subsets of nodes *within* the DHT, is described in [KR04].) By using the ReDiR library, clients can use OpenDHT to route by key among these application-specific nodes. However, because ReDiR interacts with OpenDHT only through the put/get API, the OpenDHT server-side implementation retains the simplicity of the put/get interface.

A DHT supporting multiple separate applications must distinguish them somehow; ReDiR identifies each application by an arbitrary identifier, called its *namespace*. Client nodes providing application-specific functionality join a namespace, and other clients performing lookups do so within a namespace. A ReDiR lookup on identifier *k* in namespace *n* returns the node that has joined *n* whose identifier most immediately follows *k*.

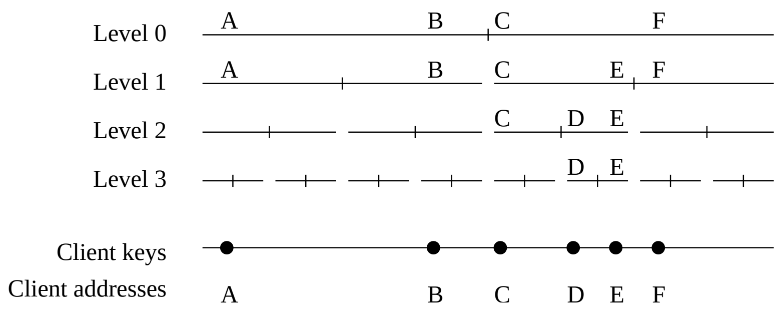


Figure 5.2: *An example ReDiR tree with branching factor $b = 2$.* Each tree node is shown as a contiguous line representing the portion of the key space covered by the node. Each node is further subdivided into two intervals separated by a tick. The names of registered application hosts (A through F) are shown above the tree nodes at which they would be stored.

A simple implementation of lookup could be achieved by storing the IP addresses and ports of all nodes that have joined a namespace *n* under key *n*; lookups could then be performed by getting all the nodes under key *n* and searching for the successor to the key looked up. This implementation, however, scales linearly in the number of nodes that join. To implement lookup more efficiently, ReDiR builds a two-dimensional quad-tree of the nodes that have joined and embeds it in OpenDHT using the put/get interface.[3] Using this tree, ReDiR performs lookup in a logarithmic number of get operations with high probability, and by estimating the tree's height based on past lookups, it reduces the average lookup to a constant number of gets, assuming client IDs are chosen uniformly at random.

The details are as follows: each tree node is list of (IP, port) pairs for a subset of the clients that have joined the namespace. An example embedding is shown in Figure 5.2. Each node in the tree has a *level*, where the root is at level 0, its immediate children are at level 1, etc. Given a branching factor of b, there are thus at most b^i nodes at level i. We label the nodes at any level from left to right, such that a pair (i, j) uniquely identifies the jth node from the left at level i, and $0 \leq j < b^i$. This tree is then embedded in OpenDHT node by node, by putting the value(s) of node (i, j) at key $H(ns, i, j)$. The root of the tree for the *i*3 application, for example, is stored at $H(\text{“i3”}, 0, 0)$. Finally, we associate with each node (i, j) in the tree b intervals of the DHT keyspace $\left[2^{160}b^{-i}(j + \frac{b'}{b}),\ 2^{160}b^{-i}(j + \frac{b'+1}{b})\right)$ for $0 \leq b' < b$.

We sketch the registration process here. Define $I(\ell, k)$ to be the (unique) interval at level ℓ that encloses key k. Starting at some level ℓ_{start} that we define later, a client with identifier v_i does

[3]The implementation of ReDiR we describe here is an improvement on our previous algorithm [KRRS04], which used a fixed tree height.

an OpenDHT get to obtain the contents of the node associated with $I(\ell_{\text{start}}, v_i)$. If after adding v_i to the list of $(IP, port)$ pairs, v_i is now the numerically lowest or highest among the keys stored in that node, the client continues up the tree towards the root, getting the contents and performing an OpenDHT put in the nodes associated with each interval $I(\ell_{\text{start}} - 1, v_i)$, $I(\ell_{\text{start}} - 2, v_i)$, …, until it reaches either the root (level 0) or a level at which v_i is not the lowest or highest in the interval. The idea is that there is one interval for each of a node's children, and the *i*th interval of a node contains only the highest and lowest entries of the entire *i*th child.

After walking up the tree in this manner, the client also walks down the tree through the tree nodes associated with the intervals $I(\ell_{\text{start}}, v_i), I(\ell_{\text{start}} + 1, v_i), \ldots$, at each step getting the current contents, and putting its address if v_i is the lowest or highest in the interval. The downward walk ends when it reaches a level in which it is the only client in the interval. Finally, since all state is soft (with TTLs of 60 seconds in our tests), the entire registration process is repeated periodically until the client leaves the system.

A lookup (ns, k) is similar. We again start at some level $\ell = \ell_{\text{start}}$. At each step we get the current interval $I(\ell, k)$ and determine where to look next as follows:

1. If there is no successor of v_i stored in the tree node associated with $I(\ell, k)$, then its successor must occur in a larger range of the keyspace, so we set $\ell \leftarrow \ell - 1$ and repeat, or fail if $\ell = 0$.
2. If k is sandwiched between two client entries in $I(\ell, k)$, then the successor must lie somewhere in $I(\ell, k)$. We set $\ell \leftarrow \ell + 1$ and repeat.
3. Otherwise, there is a client s stored in the node associated with $I(\ell, k)$ whose identifier v_s succeeds k, and there are no clients with IDs between k and v_s. Thus, v_s must be the successor of k, and the lookup is done.

A key point in our design is the choice of starting level ℓ_{start}. Initially ℓ_{start} is set to a hard-coded constant (2 in our implementation). Thereafter, for registrations, clients take ℓ_{start} to be the lowest level at which registration last completed. For lookups, clients record the levels at which the last 16 lookups completed and take ℓ_{start} to be the mode of those depths. This technique allows us to adapt to any number of client nodes while usually hitting the correct depth (Case 3 above) on the first try.

We present a performance analysis of ReDiR on PlanetLab in Section 5.4.2.

5.3 Storage Allocation

In Section 5.1.2, we presented our design goals for the OpenDHT storage allocation algorithm: that it provide storage with a definite time-to-live (TTL), that it allocate that storage fairly between clients and with high utilization, and that it avoid long periods in which no space is available for new storage requests. In this section we describe an algorithm, Fair Space-Time (FST), that meets these design goals. Before doing so, though, we first consider two choices we made while defining the storage allocation problem.

First, in this initial incarnation of OpenDHT, we equate "client" with an IP address (spoofing is prevented by TCP's three-way handshake). This technique is clearly imperfect: clients behind the same NAT or firewall compete with each other for storage, mobile clients can acquire more storage than others, and some clients (e.g., those that own class A address spaces) can acquire virtually unlimited storage. To remedy this situation, we could clearly use a more sophisticated notion of client (person, organization, etc.) and require each put to be authenticated at the gateway. However, to be completely secure against the Sybil attack [Dou02], this change would require formal identity allocation policies and mechanisms. In order to make early use of OpenDHT as easy as possible, and to prevent administrative hassles for ourselves, we chose to start with the much more primitive per-IP-address allocation model, and we hope to improve on it in the future. More generally, we discuss in Section 5.5 how our current free service could transition to a competitive commercial market in DHT service.

Second, OpenDHT is a large distributed system, and at first one might think that a fair allocation mechanism should consider the global behavior of every client (i.e., all of their puts). While tracking global behavior in this way presents a daunting problem, it is also the case that the capacity constraints of OpenDHT are per-node, in the form of finite disk capacities, so the situation is even more complicated.[4]

We note that OpenDHT cannot avoid providing some notion of per-disk fairness in allocation. For example, a common use of the system is for rendezvous, where a group of cooperating clients discover each other by putting their identities under a common key, k. With a strictly global model of fairness, a malicious client could disrupt this rendezvous by filling the disk onto which k is mapped, so long as it remained below its globally fair allocation. A per-disk model of fairness, in contrast, promises each client a fair allocation of every disk in the system, preventing such attacks.

[4]We assume that DHT load-balancing algorithms operate on longer time scales than bursty storage overloads, so their operation is orthogonal to the concerns we discuss here. Thus, in the ensuing discussion we assume that the key-to-node mapping in the DHT is constant during the allocation process.

Furthermore, the per-disk model rewards socially responsible behavior on the part of clients. Applications that are flexible in their key choice—the PAST storage system [DR01], for example—can target their puts towards otherwise underutilized nodes, thereby balancing the load on the DHT while acquiring more storage for themselves. By protecting applications that cannot choose their keys while rewarding those that can, the per-disk model reduces the need for later load balancing by the DHT itself.

For the above reasons, we have implemented per-disk fairness in OpenDHT, and we leave the study of global fairness to future work. Still, per-disk fairness is not as easy to implement as it sounds. Our storage interface involves both an amount of data (the size of the put in bytes) and a duration (the TTL). As we show below, we can use an approach inspired by fair queuing [DKS89] to allocate storage, but the two-dimensional nature of our storage requires substantial extensions beyond the original fair queuing model.

We now turn to describing the algorithmic components of FST. First we describe how to achieve high utilization for storage requests of varied sizes and TTLs while preventing starvation. Next, we introduce the mechanism by which we fairly divide storage between clients. Finally, we present an evaluation of the FST algorithm in simulation.

5.3.1 Preventing Starvation

An OpenDHT node prevents starvation by ensuring a minimal rate at which puts can be accepted at all times. Without such a requirement, OpenDHT could allocate all its storage (fairly) for an arbitrarily large TTL, and then reject all storage requests for the duration of that TTL. To avoid such situations, we first limit all TTLs to be less than T seconds and all puts to be no larger than B bytes. We then require that each OpenDHT node be able to accept at the rate $r_{min} = C/T$, where C is the capacity of the disk. We could choose a less aggressive starvation criterion, one with a smaller r_{min}, but we are presenting the most challenging case here. (It is also possible to imagine a reserved rate for future puts that is not constant over time—e.g., we could reserve a higher rate for the near future to accommodate bursts in usage—but as this change would significantly complicate our implementation, we leave it for future work.)

When considering a new put, FST must determine if accepting it will interfere with the node's ability to accept sufficiently many later puts. We illustrate this point with the example in Figure 5.3, which plots committed disk space versus time. The rate r_{min} reserved for future puts is represented by the dashed line (which has slope r_{min}). Consider two submitted puts, a large one (in

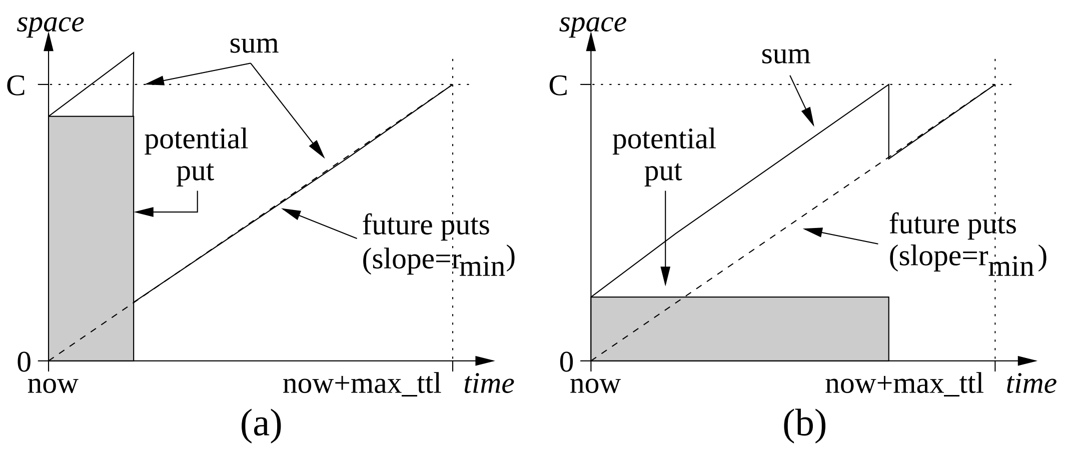


Figure 5.3: *Preventing starvation.*

terms of the number of bytes) with a short TTL in Figure 5.3(a) and a small one with a long TTL in Figure 5.3(b). The requirement that these puts not endanger the reserved minimum rate (r_{min}) for future puts is graphically equivalent to checking whether the sum of the line $y = r_{min}x$ and the top edge of the puts does not exceed the storage capacity C at any future time. We can see that the large-but-short proposed put violates the condition, whereas the small-but-long proposed put does not.

Given this graphical intuition, we derive a formal admission control test for our allocation scheme. Let $B(t)$ be the number of bytes stored in the system at time t, and let $D(t_1,t_2)$ be the number of bytes that free up in the interval $[t_1,t_2)$ due to expiring TTLs. For any point in time, call it t_{now}, we can compute as follows the total number of bytes, $f(\tau)$, stored in the system at time $t_{now}+\tau$ assuming that new puts continue to be stored at a minimum rate r_{min}:

$$f(\tau) = B(t_{now}) - D(t_{now}, t_{now}+\tau) + r_{min} \times \tau$$

The first two terms represent the currently committed storage that will still be on disk at time $t_{now}+\tau$. The third term is the minimal amount of storage that we want to ensure can be accepted between t_{now} and $t_{now}+\tau$.

Consider a new put with size x and TTL ℓ that arrives at time t_{now}. The put can be accepted if and only if the following condition holds for all $0 \le \tau \le \ell$:

$$f(\tau) + x \le C. \tag{5.1}$$

If the put is accepted, the function $f(\tau)$ is updated. As we show in Appendix A, this update can be done in time logarithmic in the number of puts accepted by tracking the inflection points of $f(\tau)$ using a balanced tree.

5.3.2 Fair Allocation

The admission control test only prevents starvation. We now address the problem of fair allocation of storage among competing clients. There are two questions we must answer in this regard: how do we measure the resources consumed by a client, and what is the fair allocation granularity?

To answer the first question, we note that a put in OpenDHT has both a size and a TTL; i.e., it consumes not just storage itself, but storage over a given time period. The resource consumed by a put is then naturally measured by the product of its size (in bytes) and its TTL. In other words, for the purposes of fairness in OpenDHT, a put of 1 byte with a TTL of 100 seconds is equivalent to a put of 100 bytes with a TTL of 1 second. We call the product of the put's size and its TTL its *commitment*.

A straightforward strawman algorithm to achieve fairness would be to track the total commitments made to each client so far, and accept puts from clients with the smallest total commitments. Unfortunately, this policy can lead to per-client starvation. To illustrate this point, assume that client A fills the disk in an otherwise quiescent system. Once the disk is full, client B begins putting its own data. B will not starve, as the admission control test guarantees that the node can still accept data at a rate of at least r_{min}, but A will starve because this strawman algorithm favors client B until it reaches the same level of total commitments granted to client A. This period of starvation could be as long as the maximum TTL, T.

To prevent such per-client starvation, we aim to equalize the *rate* of commitments (instead of the total commitments) of clients that contend for storage. Thus, the service that a client receives depends only on the competing clients at that instant of time, and not on how many commitments it was granted in the past. This strategy emulates the well known *fair queuing* algorithm that aims to provide instantaneous fairness, i.e., allocate a link capacity equally among competing flows at every instant of time.

In fact, our FST algorithm borrows substantially from the start-time fair queuing (SFQ) algorithm [GVC96]. FST maintains a system virtual time $v(t)$ that roughly represents the total commitments that a continuously active client would receive by time t. By "continuously active client" we mean a client that contends for storage at every point in time. Let p_c^i denote the i-th put of client c. Then, like SFQ, FST associates with each put p_c^i a start time $S(p_c^i)$ and a finish time $F(p_c^i)$. The start time of p_c^i is

$$S(p_c^i) = \max(v(A(p_c^i)) - \alpha, F(p_c^{i-1}), 0). \tag{5.2}$$

$A(p_c^i)$ is the arrival time of p_c^i, and α is a non-negative constant described below. The finish time of p_c^i is

$$F(p_c^i) = S(p_c^i) + \mathit{size}(p_c^i) \times \mathit{ttl}(p_c^i).$$

As with the design of any fair queuing algorithm, the key decision in FST is how to compute the system virtual time, $v(t)$. With SFQ the system virtual time is computed as the start time of the packet currently being transmitted (served). Unfortunately, in the case of FST the equivalent concept of *the* put currently being served is not well-defined since there are typically many puts stored in the system at any time t. To avoid this problem, FST computes the system virtual time $v(t)$ as the maximum start time of all puts accepted before time t.

We now briefly describe how the fairness algorithm works in conjunction with the admission control test. Each node maintains a bounded-size queue for each client with puts currently pending. When a new put arrives, if the client's queue is full, the put is rejected. Otherwise, the node computes its start time and enqueues it. Then the node selects the put with the lowest start time, breaking ties arbitrarily. Using the admission control test (Eqn. 5.1) the node checks whether it can accept this put right away. If so, the node accepts it and the process is repeated for the put with the next-lowest start time. Otherwise, the node sleeps until it can accept the pending put.

If another put arrives, the node awakes and repeats this computation. If the new put has the smallest start time of all queued puts it will preempt puts that arrived before it. This preemption is particularly important for clients that only put rarely—well below their fair rate. In such cases, the max function in Equation 5.2 is dominated by the first argument, and the α term allows the client to preempt puts from clients that are at or above their fair rate. This technique is commonly used in fair queuing to provide low latency to low-bandwidth flows [DKS89].

FST can suffer from occasional loss of utilization because of head-of-line blocking in the put queue. However, this blocking can only be of duration x/r_{min}, where x is the maximal put size, so the loss of utilization is quite small. In particular, in all of our simulations FST achieved full utilization of the disk.

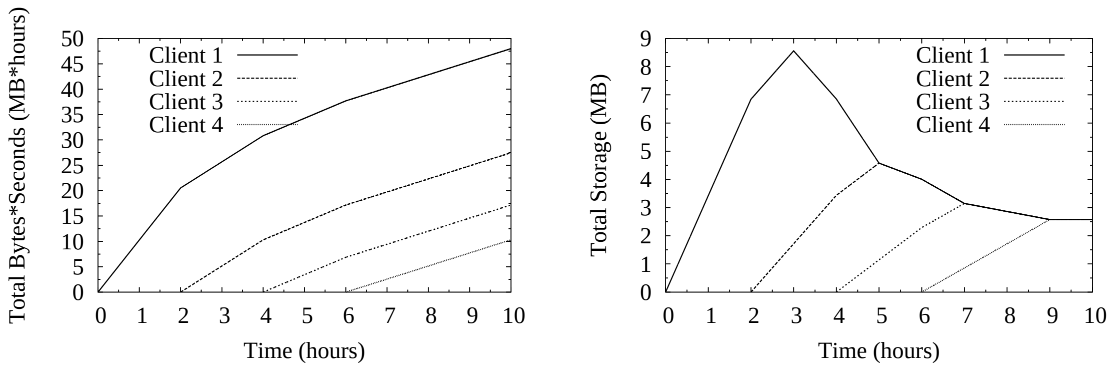


Figure 5.4: *Non-starvation.* In this experiment, all clients put above their fair rates, but begin putting at different times.

5.3.3 Evaluation

We evaluate FST according to four metrics: (1) *non-starvation*, (2) *fairness*, (3) *utilization*, and (4) *queuing latency*. We use different maximum TTL values T in our tests, but r_{min} is always 1,000 bytes per second. The maximum put size B is 1 kB. The maximum queue size and α are both set to BT.

For ease of evaluation and to avoid needlessly stressing PlanetLab, we simulate our algorithm using an event-driven simulator run on a local machine. This simulator tracks relevant features of an OpenDHT node's storage layer, but does not model any network latency or bandwidth. The interval between two puts for each client follows a Gaussian distribution with a standard deviation of 0.1 times the mean. Clients do not retry rejected puts.

Our first experiment shows that FST prevents starvation when clients start putting at different times. In this experiment, the maximum TTL is three hours, giving a disk size of 10.3 MB ($3 \times 3,600 \times 1,000$ bytes). Each client submits 1,000-byte, maximum-TTL puts at a rate of r_{min}. The first client starts putting at time zero, and the subsequent clients start putting two hours apart each. The results of the experiment are shown in Figure 5.4. The left-hand graph shows the cumulative commitments granted to each client, and the right-hand graph shows the storage allocated to each client over time.

Early in the experiment, Client 1 is the only active client, and it quickly acquires new storage. When Client 2 joins two hours later, the two share the available put rate. After three hours, Client 1 continues to have puts accepted (at $0.5r_{min}$), but its existing puts begin to expire, and its on-disk storage decreases. The important point to note here is that Client 1 is not penalized for its past commitments; its puts are still accepted at the same rate as the puts of the Client 2. While Client 1

			Test 1				Test 2				Test 3			
Client	Size	TTL	Bid	50th	90th	Avg	Bid	50th	90th	Avg	Bid	50th	90th	Avg
1	1000	60	1.0	0	974	176	2.0	5222	10851	5126	3.0	6605	12949	6482
2	1000	30	1.0	0	0	9	2.0	7248	11554	6467	3.0	7840	13561	7364
3	1000	12	1.0	0	0	9	2.0	8404	12061	7363	3.0	8612	14173	8070
4	500	60	1.0	0	409	56	2.0	7267	11551	6490	3.0	7750	13413	7368
5	200	60	1.0	0	0	13	2.0	8371	12081	7349	3.0	8566	14125	8035
6	1000	60	1.0	0	861	163	1.0	396	1494	628	1.0	446	2088	933
7	1000	30	1.0	0	0	12	1.0	237	1097	561	1.0	281	1641	872
8	1000	12	1.0	0	0	9	1.0	221	1259	604	1.0	249	1557	940
9	500	60	1.0	0	409	63	1.0	123	926	467	1.0	187	1162	770
10	200	60	1.0	0	0	14	1.0	0	828	394	1.0	6	1822	804
11	1000	60	0.5	0	768	160	0.5	398	1182	475	0.5	444	1285	531
12	1000	30	0.5	0	0	6	0.5	234	931	320	0.5	261	899	328
13	1000	12	0.5	0	0	5	0.5	214	938	306	0.5	235	891	311
14	500	60	0.5	0	288	37	0.5	137	771	226	0.5	171	825	249
15	200	60	0.5	0	0	7	0.5	0	554	103	0.5	0	715	131

Table 5.3: *Queuing times in milliseconds for each of the clients in the multiple size and TTL tests.* Sizes are in bytes; TTLs are in minutes. A “bid” of 1.0 indicates that a client is putting often enough to fill 1/15th of the disk in an otherwise idle system.

has to eventually relinquish some of its storage, the non-starvation property of the algorithm allows it to intelligently choose which data to let expire and which to renew.

As new clients arrive, the put rate is further subdivided. One maximum TTL after clients stop arriving, each client is allocated its fair share of the storage available on disk.

Our second experiment demonstrates fairness and high utilization when clients issue puts with various sizes and TTLs. In addition, it also shows that clients putting at or below their fair rate experience only slight queuing delays. The maximum TTL in this experiment is one hour, giving a disk capacity of 3.4 MB ($3,600 \times 1,000$ bytes).

We consider three tests, each consisting of 15 clients divided into three groups, as shown in Table 5.3. All the clients in a group have the same total demand, but have different put frequencies, put sizes, and TTLs; e.g., a client submitting puts with maximum size and half the maximum TTL puts twice as often as a client in the same group submitting puts with the maximum size and TTL.

The clients in Groups 2 and 3 put at the same rate in each test. The clients in Group 3 are light users. Each of these users demands only $1/30$th of the available storage. For example, Client 11 submits on average one 1,000-byte, maximum-TTL put every 30 seconds. As the fair share of each client is $1/15$th of the disk, the puts of the clients from Group 3 should be always accepted.

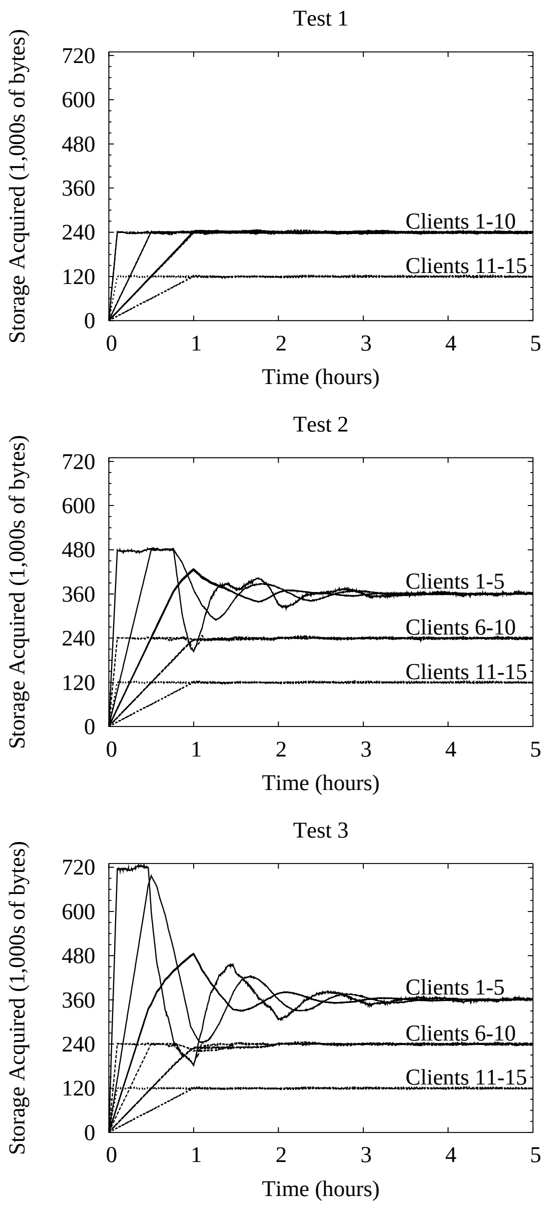


Figure 5.5: *Fair allocation despite varying put sizes and TTLs.* See text for description.

The clients in Group 2 are moderate users, putting at exactly their fair share. For example, Client 6 submits on average one 1,000-byte, maximum-TTL put every 15 seconds.

The clients in Group 1 put at a different rate in each test. In Test 1, they put as the same rate as the clients in Group 2. Since clients in Groups 1 and 2 put at their fair share while the clients in Group 3 put below their fair share, the system is underutilized in this test. In Tests 2 and 3, the clients of Group 1 put at twice and three times their fair rate, respectively. Thus, in both these tests the system is overutilized.

Figure 5.5 and Table 5.3 summarize the results for this experiment. Figure 5.5 shows the storage allocated to each client versus time. As expected, in the long term, every client receives its fair share of storage. Moreover, clients that submit puts with short TTLs acquire storage more quickly than other clients when the disk is not full yet. This effect is illustrated by the steep slopes of the lines representing the allocations of some clients at the beginning of each test. This behavior demonstrates the benefit of using the admission control test to rate-limit new put requests: looking back at Figure 5.3, one can see that many puts with short TTLs can be accepted in a mostly-empty disk without pushing the value of $f(\tau)$ over C.

Table 5.3 shows the queuing delays experienced by each client. This delay is the time a put waits from the moment it arrives at the node until it is accepted. There are three points worth noting. First, as long as the system is underutilized every client experiences very low queuing delays. This point is illustrated by Test 1.

Second, even when the system is overutilized, the clients that issue puts at below or at their fair rate experience low queuing delays. For example, the clients in Group 3 (i.e., Clients 11-15) that issue puts below their fair rate experience average queuing delays of at most 531 ms, while the clients in Group 2 (i.e., Clients 6-10) that issue puts at their fair rate experience average queuing delays no larger than 1 second. One reason clients in Group 3 experience lower queuing delays than clients in Group 2 is the use of parameter α in the computation of the start times (Eqn. 5.2). Since clients in Group 3 have fewer puts stored than those in Group 2, there are simply more cases when the start times of puts of clients in Group 3 are computed based on the system virtual time (i.e., $v(\cdot) - \alpha$) rather than on the finish times of the previous puts.

Third, clients that are above the fair rate must wait their turn more often, and thus experience higher, but not unreasonable, queuing delays.

5.4 Deployment and Evaluation

In this section we evaluate both the performance and the usability of OpenDHT.

OpenDHT's lookup and storage layers are just Bamboo's lookup and storage layers described in Chapters 3 and 4. As such, in evaluating OpenDHT's performance in Section 5.4.1, we do not focus on the detailed behavior of the underlying DHT routing or storage algorithms, both of which have been evaluated over short periods elsewhere [DLS$^+$04, RGRK04, Cat03]. Rather, we focus on the *long-running* performance of OpenDHT in terms of data durability and put/get latency. Although DHTs are theoretically capable of achieving high durability, we are aware of no previous long term studies of real (not simulated) deployments that have demonstrated this capability in practice.

As discussed in Section 5.2.2, the ReDiR library presents applications with a lookup interface. Since each ReDiR lookup is implemented using at least one get operation, a lookup in ReDiR can be no faster than a get in the underlying DHT. We quantify the performance of ReDiR lookups on PlanetLab in Section 5.4.2. This *in situ* performance evaluation is both novel (no implementation of ReDiR was offered or evaluated in [KRRS04]) and essential, as the validity of our claim that OpenDHT can efficiently support operations beyond put/get rests largely on the performance penalty of ReDiR versus standard lookup and routing interfaces.

Finally, OpenDHT's usability is best demonstrated by the spectrum of applications it supports, and we describe our early experience with these in Section 5.4.3.

5.4.1 Long-Running Put/Get Performance

In this section we report on the latency of OpenDHT gets and the durability of data stored in OpenDHT.

Measurement Setup OpenDHT has been deployed on PlanetLab since April 2004, on between 170 and 250 hosts. From August 2004 until February 2005 we continuously assessed the availability of data in OpenDHT using a synthetic put/get workload.[5] In this workload, a client puts one value into the DHT each second. Value sizes are drawn randomly from {32, 64, 128, 256, 512, 1,024} bytes, and TTLs are drawn randomly from {1 hour, 1 day, 1 week}. The same client randomly retrieves these previously put data to assess their availability; each second it randomly selects one value that should not yet have expired and gets it. If the value cannot be retrieved within an hour, a

[5]During the PlanetLab Version 3 rollout a kernel bug was introduced that caused a large number of hosts to behave erratically until it was fixed. We were unable to run OpenDHT during this period.

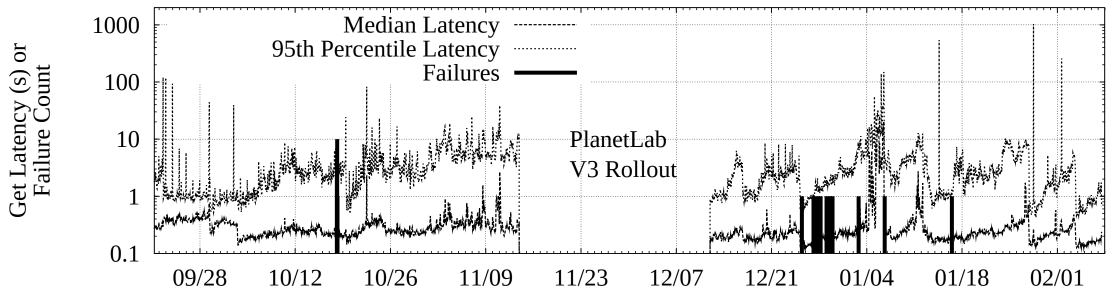


Figure 5.6: *Long-running performance and availability of OpenDHT* . See text for description.

failure is recorded. If the gateway to which the client is connected crashes, it switches to another, resubmitting any operations that were in flight at the time of the crash.

Results Figure 5.6 shows measurements taken over 3.5 months of running the above workload. We plot the median and 95th percentile latency of get operations on the *y* axis. The black impulses on the graph indicate failures. Overall, OpenDHT maintains very high durability of data; over the 3.5 months shown, the put/get test performed over 9 million puts and gets each, and it detected only 28 lost values. Get latency is underwhelming, though we show it can lowered dramatically in Chapter 7. Some of our high latency is due to bugs; on February 4 we fixed a bug that was a major source of the latency “ramps” shown in the graph. On April 22 (not shown) we fixed another and have not seen such “ramps” since. Other high latencies are caused by Internet connectivity failures; the three points where the 95th percentile latency exceeds 200 seconds are due to the gateway being partially partitioned from the Internet. For example, on January 28, the PlanetLab all-pairs-pings database [Str] shows that the number of nodes that could reach the gateway dropped from 316 to 147 for 20–40 minutes. The frequency of such failures indicates that they pose a challenge DHT designers should be working to solve; Chapter 6 describes several techniques we use to mitigate their effects.

5.4.2 ReDiR Performance

We consider three metrics in evaluating ReDiR performance: (1) latency of lookups, (2) ReDiR’s bandwidth consumption, and (3) consistency of lookups when the registered nodes external to OpenDHT churn. The first two quantify the overhead due to building ReDiR over a put/get interface, while consistency measures ReDiR’s ability to maintain correctness despite its additional level of indirection relative to DHTs such as Chord or Bamboo.

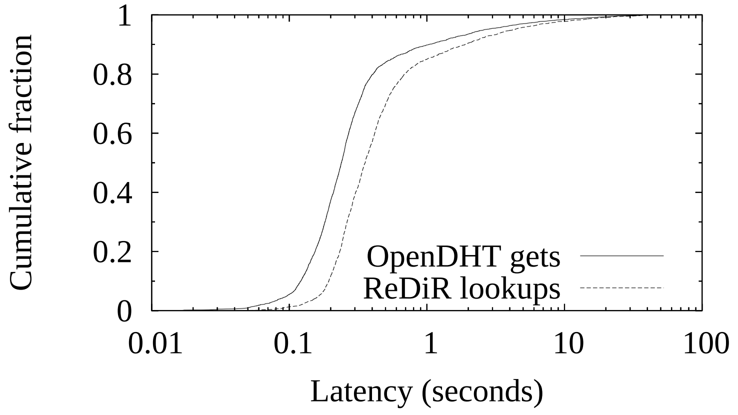


Figure 5.7: *Latency of ReDiR lookups and OpenDHT gets.*

Measurement Setup To evaluate ReDiR we had 4 PlanetLab nodes each run $n/4$ ReDiR clients for various n, with a fifth PlanetLab node performing ReDiR lookups of random keys. We selected an OpenDHT gateway for each set of clients running on a particular PlanetLab node by picking 10 random gateways from a list of all OpenDHT gateways, pinging those ten, and connecting to the one with lowest average RTT. We used a branching factor of $b = 10$ in all of our experiments, with client registration occurring every 30 seconds, and with a TTL of 60 seconds on a client's $(IP, port)$ entries in the tree. Each trial lasted 15 minutes.

Results Our first experiment measured performance with a stable set of n clients, for $n \in \{16, 32, 64, 128, 256\}$. Figure 5.7 shows a CDF of ReDiR lookup latency, based on 5 trials for each n. We compare to the latency of the OpenDHT gets performed in the process of ReDiR lookups. The average lookup uses ≈ 1.3 gets, indicating that our tree depth estimation heuristic is effective. We have verified this result in a simple simulator for up to 32,768 clients, the results of which match our PlanetLab results closely within their common range of n. Bandwidth use is quite low; even at the highest churn rate we tested, the average client registration process uses ≈ 64 bytes per second and a single lookup uses ≈ 800 bytes.

We next measured consistency as the rate of client churn varies. We used 128 clients with exponentially distributed lifetimes. Whenever one client died, a new client joined. We use the same definition of consistency used in Chapter 3; ten lookups were performed simultaneously on the same key, the majority result (if any) is considered consistent, and all others are inconsistent.

Figure 5.8 plots consistency as a function of median client lifetime. We show the mean and 95% confidence intervals based on 15 trials. Despite its layer of indirection, ReDiR is compet-

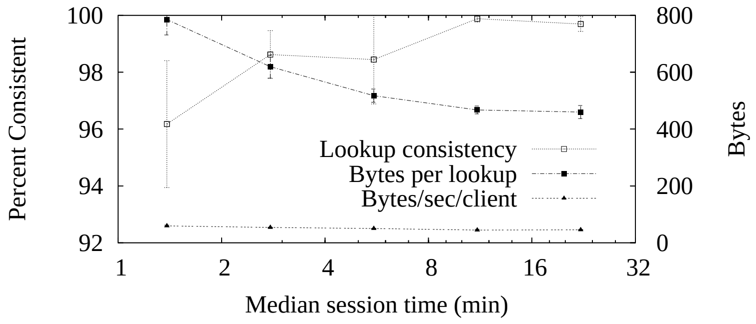


Figure 5.8: *Percentage of ReDiR lookups that are consistent, bytes transferred per lookup, and bytes/sec per registration process.*

itive with the implementation of Chord evaluated in our earlier work [RGRK04], although Bamboo performs better at high churn rates (note, however, that the experiments of [RGRK04] were performed on ModelNet, whereas ours were performed on PlanetLab).

In summary, these results show that lookup can be implemented using a DHT service with a small increase in latency, with consistency comparable to other DHTs, and with very low bandwidth.

5.4.3 Applications

We cannot directly quantify the utility of OpenDHT's interface, so in this section we instead report on our experience with building applications over OpenDHT. We first give an overview of the various OpenDHT-based applications built by us and by others. We then describe one application—FreeDB Over OpenDHT (FOOD)—in greater detail. FOOD is a DHT-based implementation of FreeDB, the widely deployed public audio-CD indexing service. As FreeDB is currently supported by a set of replicated servers, studying FOOD allows us to compare the performance of the same application built in two very different ways. We end this section with a brief discussion of common feature requests from application-builders; such requests provide one way to identify which aspects of OpenDHT matter most during development of real applications.

Generality: Overview of Applications

OpenDHT was opened up for experimentation to "friends and family" in March 2004, and to the general public in December 2004. Despite its relative infancy, OpenDHT has already been adopted by a fair number of application developers. To gain experience ourselves, we also

Application	Organization	Uses OpenDHT for . . .	put/get or ReDiR	Comments
Croquet Media Messenger	Croquet	replica location	put/get	http://opencroquet.org/
Delegation Oriented Arch. (DOA)	MIT, UCB	indexing	put/get	http://nms.lcs.mit.edu/doa/
Host Identity Protocol (HIP)	IETF WG	name resolution	put/get	alternative to DNS-based resolution
Instant Messaging Class Project	MIT	rendezvous	put/get	MIT 6.824, Spring 2004
Tetherless Computing	Waterloo	host mobility	put/get	http://mindstream.watsmore.net/
Photoshare	Jordan Middle School	HTTP redirection	put/get	http://ezshare.org/
Place Lab 802.11 Location System	IRS	location-based redirection and range queries	ReDiR	http://placelab.org/
QStream: Video Streaming	UBC	multicast tree construction	ReDiR	http://qstream.org/
RSSDHT: RSS Aggregation	SFSU	multicast tree construction	ReDiR	http://sourceforge.net/projects/rssdht/
FOOD: FreeDB Over OpenDHT	OpenDHT	storage	put/get	78 semicolons Perl
Instant Messaging Over OpenDHT	OpenDHT	rendezvous	put/get	123 semicolons C++
*i*3 Over OpenDHT	OpenDHT	redirection	ReDiR	201 semicolons Java glue between *i*3 and ReDiR, passes *i*3 regr. tests, http://i3.cs.berkeley.edu/
MOOD: Multicast Over OpenDHT	OpenDHT	multicast tree construction	ReDiR	474 semicolons Java

Table 5.4: *Applications built or under development on OpenDHT.*

developed four different OpenDHT applications. Table 5.4 lists the known OpenDHT applications. We make a number of observations:

OpenDHT put/get usage: Table 5.4 shows that the majority of these applications use only OpenDHT's put/get interface. We found that many of these (e.g., DOA, FOOD, instant messaging, HIP) make quite trivial use of the DHT—primarily straightforward indexing. Such applications are a perfect example of the benefit of a shared DHT; their relatively simple needs are trivially met by the put/get interface, but none of the applications in themselves warrant the deployment of an independent DHT.

ReDiR usage: We have four example applications that use ReDiR—two built by us and two by others. *i*3 is an indirection-based routing infrastructure built over a DHT lookup interface. To validate that ReDiR can be easily used to support applications traditionally built over a lookup interface, we ported the *i*3 code to run over OpenDHT. Doing so was extremely easy, requiring only a simple wrapper that emulated *i*3's Chord implementation and requiring *no* change to how *i*3 itself is engineered.

As described in Section 4, existing DHT-based multicast systems [CDK$^+$03a, RHKS01, ZZJ$^+$01] typically use a routing interface. To explore the feasibility of supporting such applications, we implemented and evaluated Multicast Over OpenDHT (MOOD), using a ReDiR-like hierarchy as suggested in [KRRS04]. (The QStream project has independently produced another multicast implementation based on a ReDiR-like hierarchy.) MOOD is not a simple port of an existing implementation, but a wholesale redesign. We conjecture based on this experience that one can often redesign routing-based applications to be lookup-based atop a DHT service. We believe this is an area ripe for further research, both in practice and theory.

Finally, the Place Lab project [CRR$^+$05] makes novel use of ReDiR. In Place Lab, a collection of independently operated servers processes data samples submitted by a large number of wireless client devices. Place Lab uses ReDiR to "route" an input data sample to the unique server responsible for processing that sample.

In summary, in the few months since being available to the public, OpenDHT has already been used by a healthy number of very different applications. Of course, the true test of OpenDHT's value will lie in the successful, long-term deployment of such applications; we merely offer our early experience as an encouraging indication of OpenDHT's generality and utility.

FOOD: FreeDB Over OpenDHT

FreeDB is a networked database of audio CD metadata used by many CD-player applications. The service indexes over a million CDs, and as of September 2004 was serving over four million read requests per week across ten widely dispersed mirrors.

A traditional FreeDB query proceeds in two stages over HTTP. First, the client computes a hash value for a CD—called its *discid*—and asks the server for a list of CDs with this discid. If only one CD is returned, the client retrieves the metadata for that CD from the server and the query completes. According to our measurements, this situation occurs 91% of the time. In the remaining cases, the client retrieves the metadata for each CD in the list serially until it finds an appropriate match.

A single FOOD client puts each CD's data under its discid. To query FOOD, other clients simply get all values under a discid. A proxy that translates legacy FreeDB queries to FOOD queries is only 78 semicolons of Perl.

Measurement Setup We stored a May 1, 2004 snapshot of the FreeDB database containing a total of 1.3 million discids in OpenDHT. To compare the availability of data and the latency of queries in FreeDB and FOOD, we queried both systems for a random CD every 5 seconds. Our FreeDB measurements span October 2–13, 2004, and our FOOD measurements span October 5–13.

Results During the measurement interval, FOOD offered availability superior to that of FreeDB. Only one request out of 27,255 requests to FOOD failed, where each request was tried exactly once, with a one-hour timeout. This fraction represents a 99.99% success rate, as compared with a 99.9% success rate for the most reliable FreeDB mirror, and 98.8% for the least reliable one.

In our experiment, we measured both the total latency of FreeDB queries and the latency of only the *first* HTTP request within each FreeDB query. We present this last measure as the response time FreeDB might achieve via a more optimized protocol. We consider FreeDB latencies only for the most proximal server, the USA mirror. Comparing the full legacy version of FreeDB against FOOD, we observe that over 70% of queries complete with lower latency on FOOD than on FreeDB, and that for the next longest 8% of queries, FOOD and FreeDB offer comparable response time. For the next 20% of queries, FOOD has less than a factor of two longer-latency than FreeDB. Only for the slowest 2% of queries does FOOD offer significantly greater latency than FreeDB. We attribute this longer tail to the number of request/response pairs in a FOOD transaction versus in a FreeDB transaction. Even for the idealized version of FreeDB, in which queries complete in a single HTTP GET, we observe that roughly 38% of queries complete with lower latency on FOOD than on the idealized FreeDB, and that the median 330 ms required for FOOD to retrieve all CDs' data for a discid is only moderately longer than the median 248 ms required to complete only the first step of a FreeDB lookup.

In summary, FOOD offers improved availability, with minimal development or deployment effort, and reduced latency for the majority of queries versus legacy FreeDB.

Common Feature Requests

We now briefly report experience we have gleaned in interactions with users of OpenDHT. In particular, user feature requests are one way of identifying which aspects of the design of a shared DHT service matter most during development of real applications. Requests from our users included:

XML RPC We were surprised at the number of users who requested that OpenDHT gateways accept requests over XML RPC (rather than our initial offering, Sun RPC). This request in a sense relates to generality; simple client applications are often written in scripting languages that manipulate text more conveniently than binary data structures, e.g., as is the case in Perl or Python. We have since added an XML RPC interface to OpenDHT.

Remove function After XML RPC, the ability to remove values before their TTLs expire was the most commonly requested feature in our early deployment. It was for this reason that we added remove to the current OpenDHT interface.

Authentication While OpenDHT does not currently support the immutable or signed puts we proposed in Section 5.2.1, we have had essentially no requests for such authentication from users. However, we believe this apparent lack of concern for security is most likely due to these applications being themselves in the relatively early stages of deployment.

Read-modify-write As discussed in Section 5.2.1, OpenDHT currently provides only eventual consistency. While it is possible to change values in OpenDHT by removing the old value and putting a new one, such operations can lead to periods of inconsistency. In particular, when two clients change a value simultaneously, OpenDHT may end up storing both new values. Although this situation can be fixed after the fact using application-specific conflict resolution as in Bayou [PST$^+$97], an alternate approach would be to add a read-modify-write primitive to OpenDHT. There has recently been some work in adding such primitives to DHTs using consensus algorithms [MGM05, RL03, LMR02], and we are currently investigating other primitives for improving the consistency provided by OpenDHT.

Larger maximum value size Purely as a matter of convenience, several users have requested that OpenDHT support values larger than 1 kB. OpenDHT's current 1 kB limit on values exists only due to Bamboo's use of UDP as a transport. In the near future, we plan to implement fragmentation and reassembly of data blocks in order to raise the maximum value size substantially.

5.5 Discussion

OpenDHT is currently a single infrastructure that provides storage for free. While this is appropriate for a demonstration project, it is clearly not viable for a large-scale and long-term service on which applications critically rely. Thus, we expect that any success trajectory would involve the DHT service becoming a commercial enterprise. This entails two significant changes. First, storage can no longer be free. The direct compensation may not be monetary (e.g., gmail's business model), but the service must somehow become self-sustaining. We don't speculate about the form this charging might take but only note that it will presumably involve authenticating the OpenDHT user. This could be done at the OpenDHT gateways using traditional techniques.

Second, a cooperating but competitive market must emerge, in which various competing DHT service providers (DSPs) peer together to provide a uniform DHT service, a DHT "dialtone," much as IP is a universal dialtone. Applications and clients should be able to access their DSPs (the ones to whom they've paid money or otherwise entered into a contractual relationship) and access data stored by other applications or clients who have different DSPs. We don't discuss this in detail, but a technically feasible and economically plausible peering arrangement is described by Balakrishnan et al. [BSW05]. Each DSP would have incentive to share puts and gets with other DSPs, and there are a variety of ways to keep the resulting load manageable. DHT service might be bundled with traditional ISP service (like DNS), so ISPs and DSPs would be identical, but a separate market could evolve.

If such a market emerges, then DHT service might become a natural part of the computational infrastructure on which applications could rely. This may not significantly change the landscape for large-scale, high-demand applications, which could have easily erected a DHT for their own use, but it will foster the development of smaller-scale applications for which the demand is much less certain. Our early experience suggests there are many such applications, but only time will tell.

5.6 Summary

In this chapter we have described the design and early deployment of OpenDHT, a public DHT service. Its put/get interface is easy for simple clients to use, and the ReDiR library expands the functionality of this interface so that OpenDHT can support more demanding applications. Storage is allocated fairly according to our per-IP-address and per-disk definition of fairness. The deployment experience with OpenDHT has been favorable; the system is currently supporting a variety of applications, and is slowly building a user community. The latency and availability it provides is adequate and will only get better as basic DHT technology improves.

Chapter 6

Handling Non-Transitive Connectivity

The most basic functionality of a distributed hash table, or DHT, is to partition a key space across the set of nodes in a distributed system such that all nodes agree on the partitioning. For example, as discussed throughout this thesis, Bamboo assigns each node a random identifier from the key space of integers modulo 2^{160} and maps each key k to the node whose identifier i minimizes $|i-k| \bmod 2^{160}$. So long as every Bamboo node knows its predecessor and successor in the key space, any node can compute which keys are mapped onto it.

An implicit assumption in Bamboo and other DHT protocols is that all nodes are able to communicate with each other, yet we know this assumption is unfounded in practice. We say a set of three hosts, A, B, and C exhibit *non-transitivity* if A can communicate with B, and B can communicate with C, but A cannot communicate with C. As we show in Section 6.1, 2.3% of all pairs of nodes on PlanetLab exhibit transient periods in which they cannot communicate with each other, but in which they can communicate through a third node.

Such non-transitivity in the underlying network is problematic for DHTs. Consider for example the Bamboo network illustrated in Figure 6.1. The closest node to k is B. If nodes B and C are unable to communicate with each other, however, they will both believe they are closest to k,

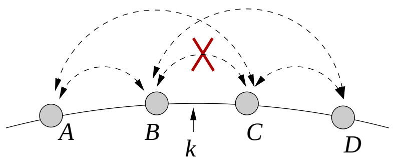


Figure 6.1: *Non-transitivity in Bamboo.* The dashed lines represent leaf set neighbor links.

and the mapping of identifiers onto nodes will not be unique.[1]

While this example may seem contrived, it is in fact quite common. If each pair of nodes with adjacent identifiers in a 300-node Bamboo network (independently) has a 0.1% chance of being unable to communicate, then we expect that there is a $1 - 0.999^{300} \approx 26\%$ chance that *some* pair will be unable to communicate at any time. However, both nodes in such a pair have a 0.999^2 chance of being able to communicate with either of the nodes that most immediately precede or follow them both.

While DHT algorithms seem quite elegant on paper, in practice we found that a great deal of the work getting Bamboo and OpenDHT to work on PlanetLab was spent discovering and fixing problems caused by non-transitivity. Of course, maintaining a full link-state routing table at each DHT node would have sufficed to solve all such problems, but would also require considerably more bandwidth than a basic DHT.[2] Instead, we discovered a set of “hacks” to cover up the false assumption of full connectivity on which DHTs are based.

After fixing Bamboo and OpenDHT so that they handle non-transitivity, we discovered that the authors of the Chord [SMK$^+$01] implementation in *i*3 [SAZ$^+$02] and the Kademlia [MM02] implementation in Coral [FFM04] had gone through a similar process.

In this chapter we report on the combined experience gleamed from these three implementation efforts. We categorize the ways in which Bamboo, Chord, and Kademlia break down under non-transitivity, and we enumerate the ways they can be modified to cope with these shortcomings. We also discuss application-level solutions to the problem.

While we present these techniques mainly for completeness of this thesis, we also hope that—at least in the short term—this work will save others the effort. In the longer term, we believe an interesting research problem is the design of a DHT algorithm that tackles non-transitivity head-on.

The next section quantifies the prevalence of non-transitivity on the Internet and surveys related work in this area. Section 6.2 presents a brief review of DHT terminology. Section 6.3 discusses four problems caused by non-transitivity in DHTs and our solutions to them. Finally, Section 6.4 concludes.

[1] This problem was first pointed out by Li et al. [LSM$^+$05], in the context of the Chord and Tapestry DHTs, although they did not present any solutions to it.

[2] For some applications, link-state routing may in fact be the right solution, but such systems are outside the scope of this thesis.

6.1 Prevalence of Non-Transitivity

The Internet is known to suffer from network outages (such as extremely heavy congestion or routing convergence problems) that result in the loss of connectivity between some pairs of nodes [Pax97, ABKM01]. Furthermore, the loss of connectivity is often non-transitive; in fact, RON [ABKM01] and SOSR [GMG$^+$02] take advantage of such non-transitivity—the fact that two nodes that cannot temporarily communicate with one another often have a third node that can communicate with them both—to improve resilience by routing around network outages.

Gerding and Stribling [GS03] observed a significant degree of non-transitivity among PlanetLab hosts; of all possible unordered three-tuples of nodes (A,B,C), about 9% exhibited non-transitivity. Furthermore, they attributed this non-transitivity to the fact that PlanetLab consists of three classes of nodes: Internet1-only, Internet2-only, and multi-homed nodes. Although Internet1-only and Internet2-only nodes cannot directly communicate, multi-homed nodes can communicate with them both. (We don't run OpenDHT on the Internet2-only nodes, so this particular form of non-transitivity does not affect us.)

Extending the above study, however, we have found that *transient* routing problems within the Internet1-only and multihomed nodes on PlanetLab are also a major source of non-transitivity. In particular, we considered a three hour window on August 3, 2005 from the all-pairs ping dataset [Str]. The dataset consists of pings between all pairs of nodes conducted every 15 minutes, with each data point averaged over ten ping attempts. We counted the number of unordered pairs of hosts (A,B) such that A and B cannot reach each other but another host C can reach both A and B. We found that, of all pairs of nodes, about 5.2% of them belonged to this category over the three hour window. Of these pairs of nodes, about 56% of the pairs had persistent problems; these were probably because of the problem described above. However, the remaining 44% of the pairs exhibited problems intermittently; in fact, about 25% of the pairs could not communicate with each other in only one of the 15-minute snapshots. This suggests that non-transitivity is not entirely an artifact of the PlanetLab testbed, but also caused by transient routing problems.

6.2 Chord and Kademlia

Before moving on to the core of this paper, we first briefly review the terminology used in Chord and Kademlia, although we assume the reader has some basic familiarity with their routing protocols. For more information, see [SMK$^+$01, MM02].

Like Bamboo, Chord and Kademlia both assign each participating node a random identifier from the key space of integers modulo 2^{160}. While Bamboo maps each key k to the node with identifier i that minimizes $|i-k| \bmod 2^{160}$, Chord maps k to the node whose identifier most immediately follows it, and Kademlia maps k to the node whose identifier i minimizes i XOR k. The root for a key in Chord is often called its *successor*.

The equivalent of Bamboo's leaf set and routing table in Chord are the *successor list*, the r nodes that most immediately follow a node, and the *finger table*, a set of nodes exponentially further away from a node around the ring. Kademlia has no direct equivalent of these structures, although it still has nearby and distant neighbors in the key space.

All three protocols greedily traverses the nodes of the DHT to perform a lookup, progressing closer to the root of the key at each step.

6.3 Problems and Solutions

This section presents problems caused by non-transitivity in DHTs and the methods we use to mitigate them. We present these problems in increasing order of how difficult they are to solve.

6.3.1 Invisible Nodes

One problem due to non-transitivity occurs when a node learns about system participants from other nodes, yet cannot directly communicate with these newly discovered nodes. This problem arises both during neighbor maintenance and while performing lookups.

For example, assume that a node A learns about a potential neighbor B through a third node C, but A and B cannot directly communicate. We say that from A's perspective B is an *invisible node*. In early versions of both Bamboo and *i*3-Chord, A would blindly add B as a neighbor. Later, A would notice that B was unreachable and remove it, but in the meantime it would try to route messages through it.

A related problem occurs when nodes blindly trust failure notifications from other nodes. Continuing the above example, when A fails to contact B due to non-transitivity, in a naive implementation A will inform C of this fact, and C will erroneously remove B as a neighbor.

A simple fix for both of these problems is to prevent nodes from blindly trusting other nodes with respect to which nodes in the network are up or down. Instead, a node A should only

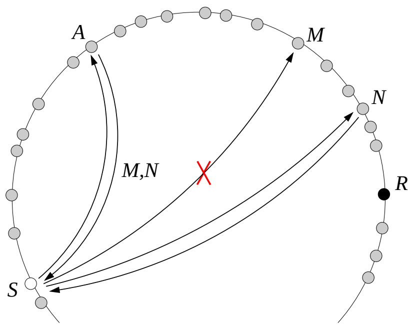


Figure 6.2: *Invisible nodes.* *S* learns about *M* and *N* from *A* while trying to route to *R*, but *S* has no direct connectivity to *M*. By sending lookup messages to *M* and *N* in parallel, *S* avoids being stalled while its request to *M* times out.

add a neighbor *B* after successfully communicating with it, and *A* should only remove a neighbor with whom it can no longer directly communicate. This technique is used by all three of our DHTs.

Invisible nodes also cause performance problems during iterative routing, where the node performing a lookup must communicate with nodes that are not its immediate neighbors in the overlay. For example, as shown in Figure 6.2, a node *S* may learn of another node *M* through its neighbor *A*, but be unable to directly communicate with *M* to perform a lookup. *S* will eventually time out its request to *M*, but such timeouts increase the latency of lookups substantially.

Three techniques can mitigate the effect of invisible nodes on lookup performance in iterative routing. First, a DHT can use virtual coordinates such as those computed by Vivaldi [CDK$^+$03b] to chose tighter timeouts. This technique should work well in general, although we have found that the Vivaldi implementations in both Bamboo and Coral are too inaccurate on PlanetLab to be of much use.

Second, a node can send several messages in parallel for each lookup, allowing requests to continue towards the root even when some others time out. As shown in Figure 6.2, *S* can send lookup messages to *M* and *N* in parallel. This technique was first proposed in Kademlia [MM02]. (We have also found it effective at reducing latency in OpenDHT; see Chapter 7.)

Third, a node can remember other nodes that it was unable to reach in the past. Using this technique, which we call *a unreachable node cache*, a node *S* marks *M* as unreachable after a few failed communication attempts. Then, if *M* is discovered again during a subsequent lookup request, *S* immediately concludes that it is unreachable without wasting bandwidth and suffering a timeout.

OpenDHT and *i*3 both use recursive routing, but Coral implements iterative routing using the above approach, maintaining three parallel RPCs and a unreachable node cache with at most

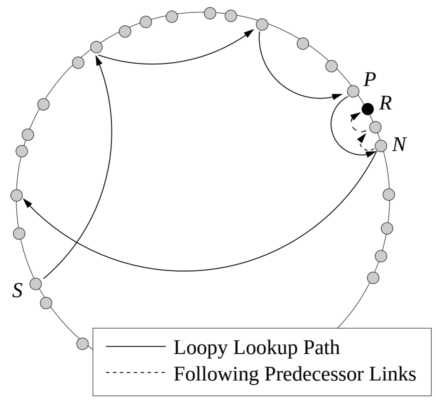


Figure 6.3: *Routing loops.* In Chord, if a lookup passes by the correct successor on account of non-transitivity, a routing loop arises. The correctness of lookup can be improved in such cases by traversing predecessor links.

512 nodes stored for at most 30 minutes each.

6.3.2 Routing Loops

In Chord, non-transitivity causes routing loops as follows. The root for a key k in Chord is the node whose identifier most immediately succeeds k in the circular key space. In Figure 6.3, let the proper root for k be R. Also, assume that P cannot communicate with R. A lookup routed through P thus skips over R to N, the next node in the key space with which P can communicate. N, however, knows its correct predecessor in the network, and therefore knows that it is not the root for k. It thus forwards the lookup around the ring, and a loop is formed.

Bamboo and Kademlia avoid routing loops by defining a total ordering over nodes during routing. In these networks, a node A only forwards a lookup on key k to another node B if $|B-k| < |A-k|$, where "$-$" represents modular subtraction in Bamboo and XOR in Kademlia.

Introducing such a total ordering in Chord is straightforward: instead of blindly forwarding a lookup towards the root, a node can stop any lookup that has already passed its root. For example, when N receives a lookup for k from P, it knows something is amiss, since $P < k < N$, but N is not the direct successor of k.

Stopping a lookup in this way avoids loops, but it is often possible to get closer to the root for a key by routing along predecessor links once normal routing has stopped. *i*3's Chord implementation backtracks in this way. For example, the dashed lines from N back to R in Figure 6.3 show the path of the lookup using predecessor links. To guarantee termination when backtracking, once a

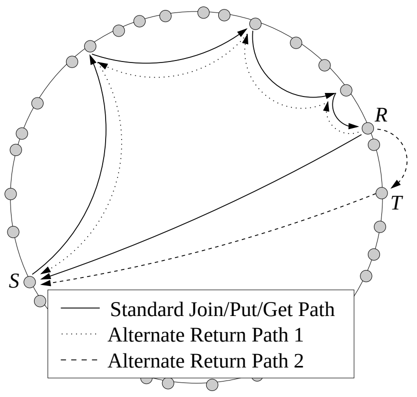


Figure 6.4: *Broken return paths.* Although *S* can route a put or get request to *R* through the overlay, there may be no direct IP route back from *R* to *S*. One alternative is to route the result back along the path taken from *S* to *R*; the other is to route through a random neighbor *T*.

packet begins following predecessor links it is never again routed along forward links. Furthermore, a time-to-live (TTL) is used to avoid long predecessor paths.

6.3.3 Broken Return Paths

Often an application built atop a DHT routing layer wants not only to route to the root of a key but also to retrieve some value back. For example, it may route a put request to the root, in which case it expects an acknowledgment of its request in return. Likewise, with a get request, it expects to receive any values stored under the given key. In one very important case, it routes a request to join the DHT to the root and expects to receive the root's leaf set or successor list in return.

As shown in Figure 6.4, when a source *S* routes a request recursively to the root *R*, the most obvious and least costly way for *R* to respond is to communicate directly with *S* (i.e., over IP). While this approach works well in the common case, it fails with non-transitivity; the existence of a route from *S* to *R* through the overlay does not guarantee the existence of the direct IP route back. We know of two solutions to this problem.

The first solution is to source route the message backwards along the path it traveled from *S* to *R* in the first place, as shown by the dotted line in Figure 6.4. Since each node along the path forwarded the message through a neighbor that had been responding to its probes for liveness, it is likely that this return path is indeed routable. A downside of this solution is that the message takes several hops to return to the client, wasting the bandwidth of multiple nodes.[3]

[3]A similar approach, where *R* uses the DHT's routing algorithm to route its response to *S*'s identifier, has a similar

A less costly solution is to have *R* source route its response to *S* through a random member of its leaf set or successor list, as shown by the dashed line in Figure 6.4. These nodes are chosen randomly with respect to *R* itself (by the random assignment of node identifiers), so most of them are likely to be able to route to *S*. Moreover, we already know that *R* can route to them, or it would not have them as neighbors.

A problem with both of these solutions is that they waste bandwidth in the common case where *R* can indeed send its response directly to *S*. To avoid this waste, we have *S* acknowledge the direct response from *R*. If *R* fails to receive an acknowledgment after some timeout, *R* source routes the response back (either along the request path or through a single neighbor). This timeout can be chosen using virtual coordinates, although we have had difficulty with Vivaldi on PlanetLab as discussed earlier. Alternatively, we can simply choose a conservative timeout value: as it is used only in the uncommon case where *R* cannot route directly to *S*, it affects the latency of only a few requests in practice.

Bamboo/OpenDHT routes back through a random leaf-set neighbor in the case of non-transitivity, using a timeout of five seconds. At the time of this writing, *i*3's Chord implementation does not handle broken return paths.

We note that iterative routing (as used in Coral) does not directly suffer from this problem. Since *S* directs the routing process itself, it will assume *R* is down and look for an alternate root, R' (the node that would be the root if *R* were actually down). Of course, depending on the application, R' may not be a suitable replacement for *R*, but that situation reduces to the inconsistent root problem, which we discuss next.

6.3.4 Inconsistent Roots

The problems we have discussed so far are all routing problems. In this section, we discuss a problem caused by non-transitivity that affects the correctness of the partitioning of the DHT key space.

Many DHT applications assume that there is only one root for a given key in the DHT at any given time. As shown in Figure 6.5, however, this assumption may be invalid in the presence of non-transitivity. In the figure, node *C* is the proper root of key *k*, but since *C* and *D* cannot communicate, *D* mistakenly believes it is the root for *k*. A lookup from S_1 finds the correct root, but a lookup from S_2 stops at *D*.

cost but a lower likelihood of success in most cases, so we ignore it here.

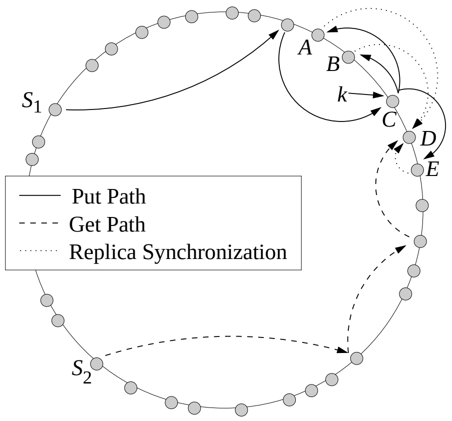


Figure 6.5: *Inconsistent roots.* A put from S_1 is routed to the root, *C*, which should replicate it on *A*–*D*. But since *C* cannot communicate with *D*, it replicates it on *A*, *B*, and *E* instead. *D* will later acquire a replica when it performs local maintenance with *A*, *B*, or *E*.

Other work has explored the issue of multiple roots due to transient conditions created by nodes joining and leaving the overlay, but did not explore the effects of misbehavior in the underlying network [CCR03b]. Furthermore, given a complete partition of the network, it is difficult to solve this problem at all, and we are not aware of any existing solutions to it. On the other hand, if the degree of non-transitivity is limited, the problem can be eliminated by the use of a consensus algorithm. The use of such algorithms in DHTs is an active area of research [MGM05, RL03, LMR02].

Nonetheless, consensus is expensive in messages and bandwidth, so many existing DHTs use a probabilistic approach to solving the problem instead. For example, FreePastry 1.4.1 maintains full link-state routing information for each leaf set, and a node is considered alive if any other member of its leaf set can route to it [fre05]. Once routability has been provided in this manner, existing techniques (e.g., [CCR03b]) can be used to provide consistency.

We note that both OpenDHT and DHash [Cat03] solve the inconsistent root problem at the application layer using the storage algorithms described in Chapter 4. As shown in Figure 6.5, OpenDHT sends a put request from S_1 for a key-value pair (k, v) to the ℓ' closest predecessors and successors of k, each of which stores a replica of (k, v). In the figure, *C* cannot communicate with *D*, and hence the wrong set of nodes store replicas.

Note that this problem will be automatically fixed by OpenDHT's replica synchronization and discard algorithms. The next time that node *D* synchronizes with node *A* or *B*, it will discover the missing value. The fate of the extra value stored by *E* depends on *E*'s connectivity. If *E* can communicate with both *C* and *D*, it will notice that it should not be storing values under key *k*, and will discard the value to one of *A*–*D*. Otherwise, it will continue storing the value, and *D* may also acquire the value from *E* in its next round of replica synchronization.

Of course, if *B* fails to synchronize with *C*–*E* between the put from S_1 and the get from S_2, it will mistakenly send an empty response for the get.

To avoid this case, for each get request on key *k*, OpenDHT queries multiple replicas for *k*, although we postpone the discussion of that technique to the next chapter.

6.4 Conclusion

In this chapter, we enumerated several ways in which Bamboo, Chord, and Kademlia break down under non-transitivity, and presented our experiences in dealing with the problems in Bamboo, as well as the related experiences of the other DHT's designers. Currently, non-transitivity is no longer a significant problem in our OpenDHT deployment. Nonetheless, it remains an interesting open problem as to whether a DHT can be designed to handle non-transitivity natively.

Chapter 7

Handling Slow Nodes

At the time of this writing, we have run OpenDHT on PlanetLab for 16 months. As described in earlier chapters, OpenDHT is built on Bamboo, which uses tested techniques to minimize latency and maximize stability. Still, our most persistent complaint from actual and potential OpenDHT users remains, “It’s just not fast enough!”

Specifically, while the long-term median latency of a get in OpenDHT is just under 200 ms—matching the best performance reported for DHash [DLS+04] on PlanetLab—the 99th percentile is measured in seconds, and even the median can rise above half a second for short periods (see Figure 5.6).

Unsurprisingly, the long tail of this distribution is caused by a few, arbitrarily slow nodes. We have observed disk reads that take tens of seconds, computations that take hundreds of times longer to perform at some times than others, and internode ping times well over a second. We are thus tempted to blame our performance woes on PlanetLab (a popular pastime in distributed systems these days), but this excuse is problematic for two reasons.

First, peer-to-peer systems are supposed to capitalize on existing resources not necessarily dedicated to the system, and do so without extensive management by trained operators. In contrast to managed cluster-based services supported by extensive advertising revenue, peer-to-peer systems were supposed to bring power to the people, even those with flaky machines.

Second, it is not clear that the problem of slow nodes is limited to PlanetLab. For example, the best DHash performance on the RON testbed, which is smaller and less loaded than PlanetLab, still shows a 99th percentile get latency of over a second [DLS+04]. Furthermore, it is well known that even in a managed cluster the distribution of individual machines’ performance is long-tailed. The performance of Google’s MapReduce system, for example, was improved by 31%

when it was modified to account for a few slow machines its designers called “stragglers” [DG04]. While PlanetLab’s performance is clearly worsened by the fact that it is heavily shared, the modern trend towards utility computing indicates that such sharing may be the case with many service infrastructures in the future.

It also seems unlikely that one could “cherry pick” a set of well-performing hosts for OpenDHT. The MapReduce designers, for example, found that a machine could suddenly become a straggler for a number of reasons, including cluster scheduling conflicts, a partially failed hard disk, or a botched automatic software upgrade. Also, as we show in Section 7.1, the set of slow nodes isn’t constant on PlanetLab or RON. For example, while the 90% of the time it takes under 10 ms to read a random 1 kB disk block on PlanetLab, over a period only 50 hours, 235 of 259 hosts will take over 500 ms to do so at least once. While one can find a set of fast nodes for a short experiment, it is nearly impossible to find such a set on which to host a long-running service.

We thus adopt the position that the best solution to the problem of slow nodes is to modify our algorithms to account for them automatically. Using a combination of delay-aware routing and a moderate amount of redundancy, our best technique reduces the median latency of get operations to 51 ms and the 99th percentile to 387 ms, a tremendous improvement over our original algorithm.

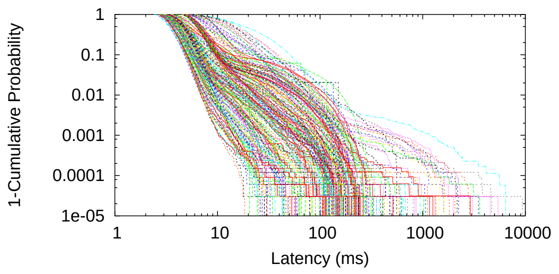
In the next section we quantify the problem of slow nodes on both PlanetLab and RON. Then, in Sections 7.2 and 7.3, we describe several algorithms for mitigating the effects of slow nodes on end-to-end get latency and show their effectiveness in an OpenDHT deployment of approximately 300 PlanetLab nodes. We conclude in Section 7.4.

7.1 The Problem of Slow Nodes

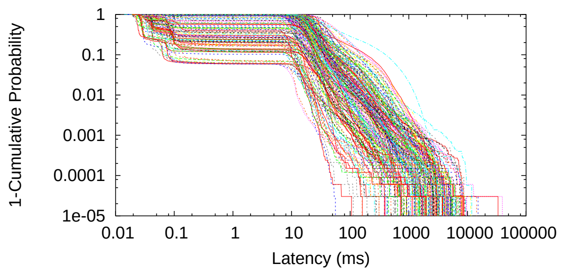
In this section, we illustrate the problem of slow nodes in PlanetLab and RON.

Figure 7.1 present CCDFs of the time to compute a 128-bit RSA key pair or read a random 1 kB block from a 1 GB file on PlanetLab using a simple C program running in its own slice. Each line represents a single node, and the lines have similar shapes. In particular, while most nodes are fast most of the time, virtually all nodes are slow some of the time, taking tens to hundreds of times longer in the worst case than the common one.

We ran the disk read test shown in Figure 7.1.b on 259 PlanetLab nodes for 50 hours, pausing five seconds between reads. Figure 7.2.a shows the number of nodes that took over 100 ms, over 500 ms, over 1 s, or over 10 s to read a block since the start of measurement. In only 6 hours, 184 nodes take over 500 ms at least once; in 50 hours, 235 do so.

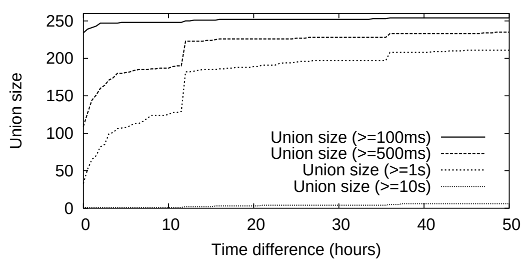


(a)

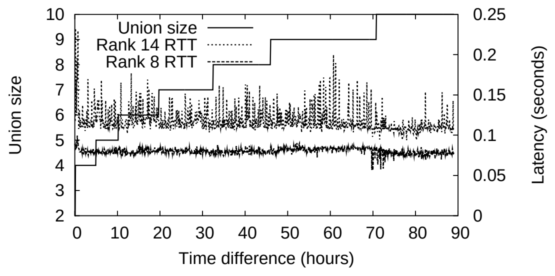


(b)

Figure 7.1: *Computation and disk read times on PlanetLab.* The left-hand graph shows the time to compute a 128-bit RSA key pair, while the right-hand graph shows the time to read a random 1 kB block from a 1 GB file.



(a)



(b)

Figure 7.2: *The variation in the set of slow nodes over time.* The left-hand graph shows the union of all PlanetLab nodes that have taken longer than 100 ms, 500 ms, 1 s, or 10 s to return a disk block since the start of the test. The right-hand graph shows the how the set of the RON nodes with the slowest network latencies to their peers changes over time.

Figure 7.2.b shows a similar graph produced from a trace of round-trip times between 15 nodes on RON [ron]. We compute for each node the median RTT to each of the other fourteen, and rank nodes by these values. The lower lines show the values for the eighth largest and second largest values over time, and the upper line shows the size of the set of nodes that have ever had the largest or second largest value. In only 90 hours, 10 of 15 nodes have been in this set. This graph shows that while the aggregate performance of the 15 nodes is relatively stable, the ordering (in terms of performance) among them changes greatly.

In summary, Figures 7.1 and 7.2 show that on both PlanetLab and RON, the slowest nodes at any time are significantly slower than those in the fastest half, and that this set of slow nodes changes relatively quickly over time.

7.2 Algorithmic Solutions

Before presenting the techniques we have used to improve get latency in OpenDHT, we give a brief overview of how gets were performed before.

7.2.1 The Basic Algorithm

Recall that the key space in Bamboo is the integers modulo 2^{160}. Each node in the system is assigned an identifier from this space uniformly at random. For fault-tolerance and availability, each key-value pair (k,v) is stored on the four nodes that immediately precede and follow k; we call these eight nodes the *replica set* for k, denoted $R(k)$. The node numerically closest to k is called its *root*.

Each node in the system knows the eight nodes that immediately precede and follow it in the key space. Also, for each (base 2) prefix of a node's identifier, it has one neighbor that shares that prefix but differs in the next bit. This latter group is chosen for network proximity; of those nodes that differ from it in the first bit, for example, a node chooses the closest from roughly half the network.

Messages between OpenDHT nodes are sent over UDP and individually acknowledged by their recipients. A congestion-control layer provides TCP-friendliness and retries dropped messages, which are detected by a failure to receive an acknowledgment within an expected time. This layer also exports to higher layers an exponentially weighted average round-trip time to each neighbor.

To put a key-value pair (k,v), a client sends a put RPC to an OpenDHT node of its choice; we call this node the *gateway* for this request. The gateway then routes a put message greedily through the network until it reaches the root for k, which forwards it to the rest of $R(k)$. When six members of this set have acknowledged it, the root sends an acknowledgment back to the gateway, and the RPC completes. Waiting for only 6 of 8 acknowledgments prevents a put from being delayed by one or two slow nodes in the replica set. These delays, churn, and Internet routing inconsistencies may all cause some replicas in the set to have values that others do not. To reconcile these differences, the nodes in each replica set periodically synchronize with each other, as described in Chapter 4.

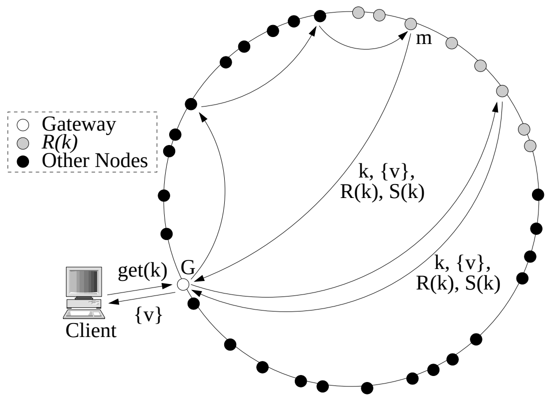


Figure 7.3: *A basic get request.*

As shown in Figure 7.3, to perform a get for key *k*, the gateway *G* routes a get request message greedily through the key space until it reaches some node $R \in R(k)$. *R* replies with any values it has with key *k*, the set $R(k)$, and the set of nodes $S(k)$ with which it has synchronized on *k* recently. *G* pretends it has received responses from *R* and the nodes in $S(k)$; if these total five or more, it sends a response to the client. Otherwise, it sends the request directly to the remaining nodes in $R(k)$ one at a time until it has at least five responses (direct or assumed due to synchronization). Finally, *G* compiles a combined response and returns it to the client.

By combining responses from at least five replicas, we ensure that even after the failure of two nodes, there is at least one node in common between the nodes that receive a put and those whose responses are used for a get.

7.2.2 Enhancements

We have explored three techniques to improve the latency of gets: delay-aware routing, parallelization of lookups, and the use of multiple gateways for each get.

Delay-Aware Routing

In the basic algorithm, we route greedily through the key space. Because each node selects its neighbors according to their response times to application-level pings, most hops are to nearby, responsive nodes. Nonetheless, a burst in load may render a once-responsive neighbor suddenly slow. Bamboo's neighbor maintenance algorithms are designed for stability of the network, and so adapt to such changes gradually. The round-trip times exported by the congestion-control layer are updated after each message acknowledgment, however, and we can use them to select among neighbors more adaptively.

The literature contains several variations on using such delay-aware routing to improve get latency. Gummadi et al. demonstrated that routing along the lowest-latency hop that makes progress in the key space can reduce end-to-end latency, although their results were based on simulations where the per-hop processing cost was ignored [GGG+03]. DHash, in contrast, uses a hybrid algorithm, choosing each hop to minimize the expected overall latency of a get, using the expected latency to a neighbor and the expected number of hops remaining in the query to scale the progress each neighbor makes in the key space [DLS+04].

We have explored several variations on this theme. For each neighbor n, we compute ℓ_n, the expected round-trip time to the neighbor, and d_n, the progress made in the key space by hopping to n, and we modified OpenDHT to choose the neighbor n with maximum $h(\ell_n, d_n)$ at each hop, where h is as follows:

Purely greedy:	$h(\ell_n, d_n) = d_n$
Purely delay-based:	$h(\ell_n, d_n) = 1/\ell_n$
Linearly scaled:	$h(\ell_n, d_n) = d_n/\ell_n$
Nonlinearly scaled:	$h(\ell_n, d_n) = d_n/f(\ell_n)$

where $f(\ell_n) = 1 + e^{(\ell_n - 100)/17.232}$. This function makes a smooth transition for ℓ_n around 100 ms, the approximate median round-trip time in the network. For round-trip times below 100 ms, the nonlinear mode thus routes greedily through the key space, and above this value it routes to minimize the per-hop delay.

Iterative Routing

Our basic algorithm performs get requests *recursively*; routing each request through the network to the appropriate replica set. In contrast, gets can also be performed *iteratively*, where the gateway contacts each node along the route path directly, as shown in Figure 7.4. While iterative requests involve more one-way network messages than recursive ones, they remain attractive because they are easy to parallelize. As first proposed in Kademlia [MM02], a gateway can maintain several outstanding RPCs concurrently, reducing the harm done by a single slow peer.

To perform a get on key k iteratively, the gateway node maintains up to p outstanding requests at any time, and all requests are timed out after five seconds. Each request contains k and the Vivaldi [CDK+03b] network coordinates of the gateway. When a node $m \notin R(k)$ receives a get request, it uses Vivaldi to compute ℓ_n relative to the gateway for each of its neighbors n, and returns

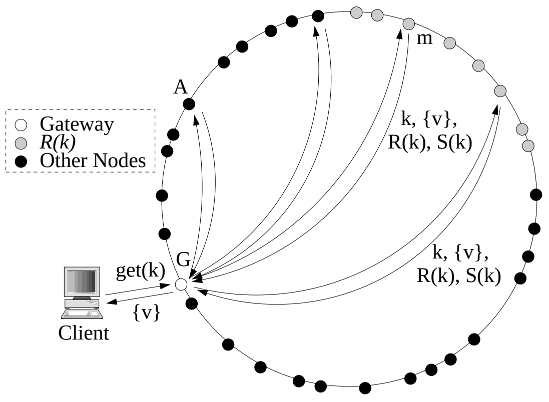


Figure 7.4: *An iterative get request.*

the three with the largest values of $h(d_n, \ell_n)$ to the gateway.

When a node $m \in R(k)$ receives a get request, it returns the same response as in recursive gets: the set of values stored under k and the sets $R(k)$ and $S(k)$. Once a gateway has received a response of this form, it proceeds as in recursive routing, collecting at least five responses before compiling a combined result to send to the client.

Multiple Gateways

Unlike iterative gets, recursive gets are not easy to parallelize. Also, in both iterative and recursive gets, the gateway itself is sometimes the slowest node involved in a request. For these reasons we have also experimented with issuing each get request simultaneously to multiple gateways. This adds parallelism to both types of get, although the paths of the get requests may overlap as they near the replica set, and it also hides the effects of slow gateways.

7.3 Experimental Results

It is well known that as a shared testbed, PlanetLab cannot be used to gather exactly reproducible results. In fact, the performance of OpenDHT varies on a hourly basis.

Despite this limitation, we were able to perform a meaningful quantitative comparison between our various techniques as follows. We modified OpenDHT such that each of the modes can be selected on a per-get basis, and we put into OpenDHT five 20-byte values under each of 3,000 random keys, re-putting them periodically so they would not expire. We then wrote a script that picks a key at random and performs one get for each possible mode in a random order. The

Parameters				Latency (ms)				Cost per Get	
GW	I/R	*p*	Mode	Avg	50th	90th	99th	Msgs	Bytes
1	Orig. Alg.			434	186	490	8113	not measured	
1	R	1	Greedy	282	149	407	4409	5.5	1833
1	R	1	Prox.	298	101	343	5192	8.7	2625
1	R	1	Linear	201	99	275	3219	6.8	2210
1	R	1	Nonlin.	185	104	263	1830	6.0	1987
1	I	3	Greedy	157	116	315	788	14.6	3834
1	I	3	Prox.	477	335	1016	2377	33.1	6971
1	I	3	Linear	210	175	422	802	18.8	4560
1	I	3	Nonlin.	230	175	455	1103	18.3	4458
1	R	1	Nonlin.	185	104	263	1830	6.0	1987
2	R	1	Nonlin.	174	99	267	1609	6.0	1987
1–2	R	1	Nonlin.	107	71	171	609	11.9	3973
1	I	3	Greedy	157	116	315	788	14.6	3834
2	I	3	Greedy	147	110	294	731	14.6	3834
1–2	I	3	Greedy	88	70	195	321	29.3	7668
1–2	I	1	Greedy	141	96	289	638	13.9	4194
1–2	I	2	Greedy	97	78	217	375	22.5	6181
1–3	R	1	Nonlin.	90	57	157	440	16.8	5332
1–4	R	1	Nonlin.	81	51	142	387	22.4	7110
1–2	I	2	Greedy	105	84	232	409	20.2	5352
1–2	I	3	Greedy	95	76	206	358	26.5	6674
1–3	I	2	Greedy	86	62	196	332	30.3	8028

Table 7.1: *Performance on PlanetLab.* *GW* is the gateway, 1–4 for planetlab(14|15|16|13).millennium.berkeley.edu. *I/R* is for iterative or recursive. The costs of the single gateway modes are estimated as half the costs of using both.

script starts each get right after the previous one completes, or after a timeout of 120 seconds. After trying each mode, the script picks a new key, a new random ordering of the modes, and repeats. So that we could also measure the cost of each technique, we further modified the OpenDHT code to record the how many messages and bytes it sends on behalf of each type of get. We ran this script from July 29, 2005 until August 3, 2005, collecting 27,046 samples per mode to ensure that our results cover a significant range of conditions on PlanetLab.

Table 7.1 summarizes the results of our experiments.

The first row of the table shows that our original algorithm, which always routed all the way to the root, takes 186 ms on median and over 8 s at the 99th percentile.

The next block of four rows shows the performance of the basic recursive algorithm of

Section 7.2.1, using only one gateway and each of the four routing modes described in Section 7.2.2. We note that while routing with respect to delay alone improves get latency some at the lower percentiles, the linear and nonlinear scaling modes greatly improve latency at the higher percentiles as well. The message counts show that routing only by delay takes the most hops, and with each hop comes the possibility of landing on a newly slow node; the scaled modes, in contrast, pay enough attention to delays to avoid the slowest nodes, but still make quick progress in the key space.

We note that the median latencies achieved by all modes other than greedy routing are lower than the median network RTT between OpenDHT nodes, which is approximately 137 ms. This seemingly surprising result is actually expected; with eight replicas per value, the DHT has the opportunity to find the closest of eight nodes on each get. Using the distribution of RTTs between nodes in OpenDHT, we computed that an optimal DHT that magically chose the closest replica and retrieved it in a single RTT with no processing delay would have a median get latency of 31 ms, a 90th percentile of 76 ms, and a 99th percentile of 130 ms.

The next four rows show the same four modes, but using iterative routing with a parallelism factor, p, of 3. Note that the non-greedy modes are not as effective here as for recursive routing. We believe there are two reasons for this effect. First, the per-hop cost in iterative routing is higher than in recursive, as each hop involves a full round-trip, and on average the non-greedy modes take more hops for each get. Second, recursive routing uses fresh, direct measurements of each neighbor's latency, but the Vivaldi algorithm used in iterative routing cannot adapt as quickly to short bursts in latency due to load.

Despite their inability to capitalize on delay-awareness, the extra parallelism of iterative gets provides enough resilience to far outperform recursive ones at the 99th percentile. This speedup comes at the price of a factor of two in bandwidth used, however.

The next three rows show the benefits of using two gateways with recursive gets. We note that while both gateways are equally slow individually, waiting for only the quickest of them to return for any particular get greatly reduces latency. In fact, for the same cost in bandwidth, they far outperform iterative gets at all percentiles.

The next three rows show that using two gateways also improves the performance of iterative gets, reducing the 99th percentile to an amazing 321 ms, but this performance comes at a cost of roughly four times that of recursive gets with a single gateway.

The next two rows show that we can reduce this cost by reducing the parallelism factor, p, while still using two gateways. Using $p = 1$ gives longer latencies than recursive gets with the same cost, but using $p = 2$ provides close to the performance of $p = 3$ at only three times the cost of recursive gets with a single gateway.

Since iterative gets with two gateways and $p = 3$ use more bandwidth than any of the recursive modes, we ran a second experiment using up to four gateways per get request. This experiment involved 80,000 samples per mode collected from August 3, 2005 until August 9, 2005. The final five rows show the results. For the same cost, recursive gets are faster than iterative ones at both the median and 90th percentile, but slower at the 99th.

These differences make sense as follows. As the gateways are co-located, we expect the paths of recursive gets to converge to the same replica much of the time. In the common case, that replica is both fast and synchronized with its peers, and recursive gets are faster, as they have more accurate information than iterative gets about which neighbor is fastest at each hop. In contrast, iterative gets with $p > 1$ actively explore several replicas in parallel and are thus faster when one discovered replica is slow or when the first replica is not synchronized with its peers, necessitating that the gateway contact multiple replicas.

7.4 Conclusions

In this chapter we highlighted the problem of slow nodes in distributed systems, and we demonstrated that their effect on overall system performance can be mitigated through a combination of delay-aware algorithms and a moderate amount of redundancy. Using only delay-awareness, we reduced the 99th percentile get latency from over 8 s to under 2 s. Using a factor of four more bandwidth, we can further reduce the 99th percentile to under 400 ms and cut the median by a factor of three.

Looking beyond our specific results, we note that there has been a lot of collective hand-wringing recently about the value of PlanetLab as an experimental platform. The load is so high, it is said, that one can neither get high performance from an experimental service nor learn interesting systems lessons applicable elsewhere.

We have certainly cast some doubt on the first of these two claims. The latencies shown in Table 7.1 are low enough to enable many applications that were once thought to be outside the capabilities of a “vanilla” DHT. For example, Cox et al. [CMM02] worried that Chord could not be used to replace DNS, and others argued that aggressive caching was required for DHTs to do so [RS04a]. In contrast, even our least expensive modes are as fast as DNS, which has a median latency of around 100 ms and a 90th percentile latency of around 500 ms [JSBM01].

As to the second claim, there is no doubt that PlanetLab is a trying environment on which to test distributed systems. That said, we suspect that the MapReduce designers might say the same about their managed cluster. Their work with stragglers certainly bears some resemblance to the problems we have dealt with. While the question is by no means settled, we suspect that PlanetLab may differ from their environment mainly by degree, forcing us to solve problems at a scale of 300 nodes that we would eventually have to solve at a scale of tens of thousands of nodes. If this is the case, perhaps PlanetLab's slowness is not a bug, but a feature.

Chapter 8

Conclusion

The unavoidable price of reliability is simplicity.
— C.A.R. Hoare

In this thesis we have presented the Bamboo DHT and the OpenDHT service. The Bamboo lookup layer supports low-latency lookups under very high churn rates; with session times as short as six minutes, a 1,000-node Bamboo network on ModelNet is still able to average around one half second per lookup. The Bamboo storage layer supports reliable, high-performance put/get and remove operations. Running on 200–300 nodes on PlanetLab, it has provided very high availability as measured over months, and it maintains very low get latencies despite the presence of arbitrarily slow nodes. Both the lookup and storage layers in Bamboo are resilient to non-transitivity in the underlying network, a requirement for long-term use in real deployments. Furthermore, Bamboo is a complete implementation; no part of the system is only run in simulation.

We have also presented OpenDHT, a public DHT service designed to ease the deployment and maintenance of DHT-based applications. By providing an existing DHT deployment with a simple put/get interface over RPC, OpenDHT allows the construction of DHT applications in tens of lines of code.

The current put/get interface to OpenDHT is secure against most attacks. Puts cannot be removed by arbitrary clients of the system, but only by those who know a secret chosen at the time of the put. The particular technique used is secure against packet sniffing as well. While the current interface is still vulnerable against drowning attacks, where malicious clients bury an important value in other ones, a planned interface using public-key cryptography is not.

OpenDHT does not limit more sophisticated DHT applications to the put/get interface.

The ReDiR library and its variants have been used to implement lookup, multicast, and range-search. In this capacity OpenDHT serves as a common rendezvous point for many distinct applications.

OpenDHT also guarantees a fair share of storage to each client in the system using its fair space-time (FST) algorithm. While this algorithm is not yet deployed in the production system, simulations show that it can provide fair shares of storage without causing starvation.

Except for the public-key interface and FST, OpenDHT is also a complete implementation. It has been running continuously on PlanetLab and available for public use since April 2004. Two ReDiR implementations are available, as are implementations of multicast and range search.

In the remainder of this chapter we look back on the design decisions we made throughout the course of this research and assess them with the benefit of hindsight.

Simplicity and Reliability

We have opened this chapter with a quote by C.A.R. Hoare, "The unavoidable price of reliability is simplicity." Throughout our work on Bamboo and OpenDHT, we have found this to be the best piece of advice one could give to a distributed systems researcher. In fact, it never ceases to amaze us that a mere 3,000 lines of code (the size of the Bamboo router) can behave so unpredictably in aggregate when run simultaneously on 1,000 nodes.

The Pastry Partitioning

The choice of the Pastry algorithm's partitioning scheme, that each key k is mapped to the node whose identifier i minimizes $|i-k| \bmod 2^{160}$, was unwittingly wise. Using this metric, a node can examine at any lookup message, and without knowing where it came from or where it has been, decide whether the local node is the root or whether it should to forward the lookup on. We cannot overemphasize the degree to which this makes programming the system easier. Because of this simplicity, we were able to get a basic version of the Bamboo router working in under a week, and it later continued to serve us well as we dealt, for example, with non-transitivity in the underlying network.

Epidemic Algorithms

The use of epidemic algorithms in both the routing neighbor maintenance algorithms and the storage management algorithms has also greatly simplified the code. We are glad to have stumbled across the epidemic literature when we did; our first rewrite of the storage maintenance algorithms using epidemic techniques required less bandwidth during node failures and joins, was less buggy, and used less code than its predecessor. The combination of using a DHT to partition the key space and epidemics to maintain consistency within a portion of that space has proved a fruitful one.

From one point of view, we think of epidemic algorithms as embodying the philosophy that in trying to get things exactly right, one is just as likely to mess everything up. Instead, one should at each step simply try and make things a little better. For example, each time we move a replica in the storage layer, we're not necessarily placing it onto all of the "right" nodes, but we are moving it to at least one node that should have it and does not. This point is subtle, but keeping this philosophical notion in mind has greatly eased the design of most of the algorithms used in Bamboo.

Link Structure

The choice of routing table neighbors in Bamboo, in retrospect, seems rather arbitrary to us now. Gummadi et al. [GGG+03] have shown that it has few (if any) benefits over other $O(\log N)$ link structures (e.g., Chord), and this observation is born out by our experience. A more adaptive structure, perhaps Accordion [LSMK05], may be the wave of the future. As Accordion has not yet been fully implemented and "battle tested" on PlanetLab, however, it is probably too early to say for sure.

The Pastry Routing Algorithm

Like the Pastry link structure, we're not sure of the value of the Pastry routing algorithm, either. If the utmost simplicity is the goal, simple greedy routing through the key space is easier to explain, and for minimal route latency, the non-linear scaling of Chapter 7 is superior. The right interface to a DHT router would probably allow each application to choose what routing metric it preferred; the latest Bamboo code does just that.

Put/Get/Remove

The simplicity of put/get/remove, especially our implementations of put as append, get as iterate, and secure, value-specific remove seems to have been a wise choice. Its semantics are easy to understand, and it is easy for clients outside the DHT to observe what operations have been performed and make application-level decisions if necessary. If an application prefers that only one value be stored per key, it can create a total ordering over all values (by including a sequence number and client IP address with each value, for example) and remove the lower-ordered values when conflicts occur. But we prefer not to force such semantics on all applications.

Not “Cherry-Picking” Nodes

Whenever we have had a problem with a particular PlanetLab node, we have been tempted to simply stop running OpenDHT on it. Instead, we have tried in each case to find some way to have the system automatically work around it. The work in Chapter 7, for example, grew out of this philosophy. In the long run, this approach makes the system more robust and greatly reduces the amount of time we spend maintaining the system. As a result, as of this writing we have not touched the maintenance interface to the public deployment in over three weeks!

Dataflow Architectures

A recent bit of research that has inspired us is the dataflow-based overlay work of Loo et al. [LCH+05]. While it is still in its early stages, the dataflow approach and possibly even the declarative specification of overlay architectures seem promising to us.

Java

Despite the doubts raised about it by many, Java has proven to be a perfectly reasonable distributed systems programming language. It is much faster to prototype in than C or C++, and since most of the time the Bamboo/OpenDHT code is either waiting on the disk or the network, its higher-level features have little effect on performance. Garbage collection can still cause annoying pauses in execution from time to time, although these have improved with the introduction of incremental collectors, and should be further reduced in the future with the continuing development of concurrent collectors. Moreover, as the code is already designed to handle arbitrarily slow nodes, garbage collection can be seen as just another source of unpredictable slowness.

The Nature of Systems Research

As a final point, we would like to comment on the nature of systems research and our place in it. As we see it, there are three main types of systems research. In the first type, the so called measurement study, one analyzes an existing system to discover features of its workload that might be used to build a better system in the future. In the second type, what we will call the design study, one takes the results of some measurement study and proposes a new system; this research is usually continued to the point of building a prototype and demonstrating its improved performance through simulation. In the third type of systems research, one builds a complete implementation of a system that one has already motivated, prototyped, simulated, and published papers about, simply because one believes that (1) it will be useful to others as an artifact and (2) one will learn more by actually building it. As should be clear from our tone, while we see all three types of systems research as valuable in their own right, it is this third type that excites us most. While it can be painful at times—the Bamboo [RGRK04] and OpenDHT [RGK$^+$05] papers, for example, were each rejected twice from top conferences before being accepted—it is also very rewarding. We cannot overstate the simple joy of a successful demo, for example.

Appendix A

The Storage Tree

Recall from Chapter 5 that for any time t_{now}, we can produce a function, $f(\tau)$, which represents the expected number of bytes in the system at a future time $t_{now}+\tau$, assuming that new puts continue to be stored at a minimum rate r_{min}:

$$f(\tau) = B(t_{now}) - D(t_{now}, t_{now}+\tau) + r_{min} \times \tau$$

The first two terms represent the currently committed storage that will still be on disk at time $t_{now}+\tau$. The third term is the minimal amount of storage that we want to ensure can be accepted between t_{now} and $t_{now}+\tau$. A new put with size x and TTL ℓ that arrives at time t_{now} can be accepted if and only if the following condition holds for all $0 \leq \tau \leq \ell$:

$$f(\tau) + x \leq C.$$

If the put is accepted, the function $f(\tau)$ must be updated, otherwise, we would like to compute at what time in the future we will be able to accept the put. In this appendix we justify the claim from Chapter 5 that these computations can be performed in time logarithmic in the number of puts in the system at any time.

A.1 The Storage Tree

Our technique is to build a tree whose leaves represent the inflection points of $f(\tau)$. This data structure has three primary functions:

- shiftTime$(n, t_{now}, r_{min}, t'_{now})$ takes the tree n that represents $f(\tau)$ over the time period $[t_{now}, t_{now} + T]$ and returns a new tree that represents $f(\tau)$ over the time period $[t'_{now}, t'_{now} + T]$, where T is the maximum TTL in the system and $t'_{now} \geq t_{now}$.

- nextAccept$(n, t_{now}, r_{min}, C, x, \ell)$ takes the tree n that represents $f(\tau)$ over the time period $[t_{now}, t_{now} + T]$ and returns at what time in the future we will be able to accept a put of size x and TTL ℓ, given the maximum size of the disk, C.

- addPut$(n, t_{now}, r_{min}, \ell, x)$ takes the tree n that represents $f(\tau)$ over the time period $[t_{now}, t_{now} + T]$ and returns a new tree over the same time period that also encompasses a put of size x and TTL ℓ accepted at time t_{now}.

Given these functions, the FST algorithm is as described in Chapter 5: we compute which put we are going to accept next based on fair sharing concerns, shift time in the storage tree to the current time, compute at what future time we can accept the put, sleep until that time, shift the storage tree's time once more, add the put to the tree, send an accept message to the client, and loop.

A.2 The Basic Data Structure

Each node in the tree has eight fields: $\mathit{offset}, \mathit{value}, \mathit{low}, \mathit{high}, \mathit{height}, \mathit{valid}, \mathit{left}$, and right. The meaning of a node n is that the maximum value of the function $f(\tau)$ in the range $[n.\mathit{low}, n.\mathit{high}]$ is equal to $n.\mathit{value} + n.\mathit{offset}$ plus the sum of the offsets of n's parents in the tree. In this way, the sum $\mathit{value} + \mathit{offset}$ at the root of the tree is equal to the maximum value of $f(\tau)$ for all $0 \leq \tau \leq T$.

The tree is implemented as follows. We use **nil** to represent an empty tree—in other words, a tree that represents no puts. Otherwise, every node in the tree is either a leaf or has two children. In a leaf, it is always the case that

- $\mathit{left} = \mathit{right} = \textbf{nil}$

- $\mathit{low} = \mathit{high}$

- $\mathit{height} = 1$

- $\mathit{value} = 0$

In other words, leaves have no children, they are always at the lowest level of the tree, and they represent a single point in time, *low*. The *valid* flag is used to remove nodes from the tree as described below.

In an interior node, *left* and *right* are the children, and it is always the case that

- $low = left.low$
- $high = right.high$
- $height = 1 + \max(left.height, right.height)$
- $value = \max(left.valid ? left.offset + left.value : 0, right.valid ? right.offset + right.value : 0)$

In other words, an interior node represents the combined time range of its children, and its height is one greater than the height of the highest child. The *value* of an interior node represents the maximum $value + offset$ of its *valid* children, or zero if both children are invalid.

All procedures that operate on the storage tree are functional in nature; once created, a node in the tree can never be changed. We define the following constructors:

- $\text{makeLeaf}(offset, t)$ — creates a leaf with the given offset, $high = low = t$, and $valid = \textbf{true}$.
- $\text{makeParent}(offset, left, right)$ — creates an interior node with the given offset and children, which satisfies the restrictions on interior nodes specified above, and where $valid = left.valid \vee right.valid$.
- $\text{incrementOffset}(n, i)$ — creates a node n' identical to n except that $n'.offset = n.offset + i$.
- $\text{invalidate}(n)$ — creates a node n' identical to n except that $n'.valid = \textbf{false}$.

There are no other node constructors.

A.3 The Storage Tree Functions

In order to ensure that all operations on the storage tree are efficient, we maintain the invariant that the tree is balanced. Specifically, for every interior node we ensure that

$$|left.height - right.height| \leq 1.$$

For convenience, we define for each node a virtual field, *balanced*, that is true if and only if the above condition holds (and that is always true for a leaf node). To balance a (sub)tree that has become unbalanced due to an insertion, we use Algorithms 1–3. These functions are no more than the usual balancing functions for an AVL tree, except that they are extended to respect the semantics of the *offset*s while rotating. In particular, as in an AVL tree, using these balancing functions does not effect the logarithmic cost of an insertion into the tree.

We next introduce three auxiliary functions as Algorithms 4–6. createPoint(n, r_{min}, t) adds a new leaf at time t to an existing tree n. The value of f represented by the tree is unchanged. We use createPoint, for example, when we want to increment or decrement f over a range of times and we want to make sure that the endpoints of this range are represented as leaves in the tree. Except that it maintains the semantics of *offset*, *valid*, etc. in the tree, createPoint is identical to the insert function of an AVL tree, and thus has a logarithmic running time.

Our next auxiliary function is incrementRange(n, l, h, i), which increments the *offset*s on the minimal set of nodes that cover only the range of times $[l, h]$. In particular, the tree n must already have leaves with times l and h for incrementRange to succeed. incrementRange follows at most two paths down and back up the tree: one corresponding to time l and the other to time h, and thus has a logarithmic running time as well.

Our final auxiliary function is invalidateRange(n, l, h, i), which is identical to incrementRange except that rather than modifying the *offset* fields of nodes, it sets their *valid* flags to **false**.

We are now ready to introduce addPut and shiftTime as Algorithms 7–8. As described above, addPut$(n, t_{now}, r_{min}, \ell, x)$ adds a put of size x and TTL ℓ to the tree n. The code for addPut is best understood visually, as illustrated in Figure A.1. The three calls to either makeLeaf or createPoint are creating the three inflection points needed to represent a put with size x and TTL ℓ. In the case where the tree was initially empty, we just add parents over these leaves in a balanced way, and in the case where we have an existing tree, we just call incrementRange along the length of the put after creating the necessary points. addPut does not loop, and it simply calls several logarithmic-time functions in sequence, so it takes logarithmic time itself.

As described above, shiftTime$(n, t_{now}, r_{min}, t'_{now})$ shifts the time represented by tree n from $[t_{now}, t_{now} + T]$ to $[t'_{now}, t'_{now} + T]$. First, it creates a point for t'_{now}. Then, it invalidates all nodes before t'_{now}. Finally, it decrements the value of f by $r_{min}(t'_{now} - t_{now})$. The reason for this latter operation is best understood visually; looking again at Figure A.1, we are moving the line $y = r_{min}x$ (which represents expected future puts) from its current intersection with the x axis at 0 to a new intersection at $t'_{now} - t_{now}$.

One problem with shiftTime as specified is that it never removes any nodes from the tree. How, then, does the tree ever get smaller? As discussed above, createPoint is little more than an AVL tree insert operation. Since it inserts only one point, it disrupts the tree's balance by at most one, and

Algorithm 1 Rotate right.

```
rotateRight($n$)
  return makeParent($n.offset$,
          incrementOffset($n.left.left, n.left.offset$),
          makeParent($0$, incrementOffset($n.left.right, n.left.offset$), $n.right$))
```

Algorithm 2 Rotate left.

```
rotateLeft($n$)
  return makeParent($n.offset$,
          makeParent($0, n.left$, incrementOffset($n.right.left, n.right.offset$)),
          incrementOffset($n.right.right, n.right.offset$))
```

Algorithm 3 Balance the tree.

```
balance($n$)
  if $n = \mathbf{nil} \vee n.left = \mathbf{nil} \vee n.balanced$ then
    return $n$
  else
    if $n.left.height > n.right.height$ then
      if $n.left.left.height > n.left.right.height$ then
        return rotateRight($n$)
      else
        return rotateRight(makeParent($n.offset$, rotateLeft($n.left$), $n.right$))
      end if
    else
      if $n.right.right.height > n.right.left.height$ then
        return rotateLeft($n$)
      else
        return rotateLeft(makeParent($n.offset, n.left$, rotateRight($n.right$)))
      end if
    end if
  end if
```

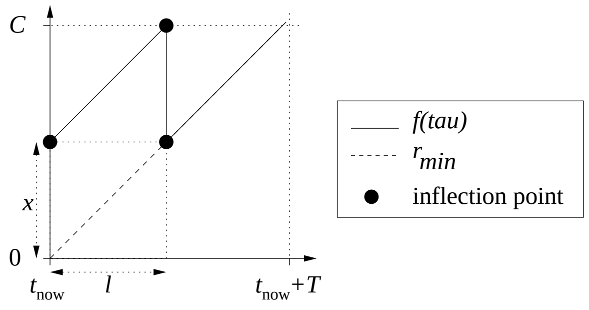


Figure A.1: addPut visualized.

Algorithm 4 Add a new inflection point to the tree.

```
createPoint($n, r_{min}, t$)
  if $n.left = $ nil then
    if $n.low = t$ then
      return $n$
    else if $t < n.low$ then
      return makeParent($0$, makeLeaf($n.offset + n.r_{min}(n.low - t), t$), $n$)
    else
      return makeParent($0, n$, makeLeaf($n.offset + n.r_{min}(t - n.low), t$))
    end if
  else
    if $t \leq n.left.high$ then
      return balance(makeParent($n.offset$, balance(createPoint($n.left, r_{min}, t$)), $n.right$))
    else
      return balance(makeParent($n.offset, n.left$, balance(createPoint($n.right, r_{min}, t$))))
    end if
  end if
```

Algorithm 5 Increment the offsets for a range of time.

```
incrementRange($n, l, h, i$)
  if $l > n.high \vee h < n.low$ then
    return $n$
  else if $l \leq n.low \wedge h \geq n.high$ then
    return incrementOffset($n, i$)
  else
    return makeParent($n.offset$, incrementRange($n.left, l, h, i$), incrementRange($n.right, l, h, i$))
  end if
```

Algorithm 6 Invalidate a range of time.

```
invalidateRange($n, l, h$)
  if $l > n.high \vee h < n.low$ then
    return $n$
  else if $l \leq n.low \wedge h \geq n.high$ then
    return invalidate($n$)
  else
    return makeParent($n.offset$, invalidateRange($n.left, l, h$), invalidateRange($n.right, l, h$))
  end if
```

therefore it can be rebalanced just as in an AVL tree. AVL trees also have a remove function that keeps the tree balanced, but shiftTime invalidates many nodes at once. As such, were it to remove the nodes directly, it might create a tree with an imbalance greater than 1 (i.e., the left subtree of a node might have a *height* two greater than the right subtree). To circumvent this problem, instead of removing nodes from the tree, we merely mark them as not *valid* in shiftTime. Then, whenever the entire left subtree of the root is not *valid*, we remove it and promote the right subtree to the root. To do this, we introduce the shiftTimeRoot function, shown as Algorithm 9, which is always used instead of shiftTime when shifting the time covered by the root of the tree.

We are finally ready to introduce our last top-level function for the storage tree, nextAccept, as Algorithm 10. As described above, nextAccept$(n, t_{now}, r_{min}, C, x, \ell)$ returns at what time in the future we will be able to accept a put of size x and TTL ℓ. nextAccept is implemented as binary search of the times between t and $t + x/r_{min}$. (We know we can accept the put at time $t + x/r_{min}$.)

Since nextAccept calls addPut and shiftTimeRoot $O(\log m)$ times, where m is the maximum value of x/r_{min}, the running time of nextAccept is $O(\log m \log p)$, where p is the number of puts stored in the system. However, since m is also the maximum time the put at the head of the queue will wait before being accepted, there is a desire to keep m small. In OpenDHT, we do this by bounding the maximum put size. If we also bound the minimum rate r_{min}, then m is also bounded, and hence the running time of nextAccept is merely logarithmic in the number of puts accepted.

A.4 Status

The storage tree is implemented as described in this chapter and was used for the FST simulations in Chapter 5. It is not yet deployed on PlanetLab, however.

Algorithm 7 Add a put to the tree.

```
addPut($(n, t_{now}, r_{min}, \ell, x)$)
   **if** $n =$ **nil then**
      $a \leftarrow$ makeLeaf$(x, t_{now})$
      $b \leftarrow$ makeLeaf$(x + r_{min}(\ell - 1), t_{now} + \ell - 1)$
      $c \leftarrow$ makeParent$(0, a, b)$
      $d \leftarrow$ makeLeaf$(r_{min}\ell, t_{now} + \ell)$
      **return** makeParent$(0, c, d)$
   **else**
      $a \leftarrow$ createPoint$(n, r_{min}, t_{now})$
      $b \leftarrow$ createPoint$(a, r_{min}, t_{now} + \ell - 1)$
      $c \leftarrow$ createPoint$(b, r_{min}, t_{now} + \ell)$
      **return** incrementRange$(n, t_{now}, t_{now} + \ell - 1, x)$
   **end if**
```

Algorithm 8 Shift the tree to represent only nodes from t'_{now} onwards.

```
shiftTime$(n, t_{now}, r_{min}, t'_{now})$
   **if** $n =$ **nil then**
      **return** $n$
   **else**
      $a \leftarrow$ createPoint$(n, r_{min}, t'_{now})$
      $b \leftarrow$ invalidateRange$(n, t_{now}, t'_{now} - 1)$
      $c \leftarrow$ incrementRange$(b, t'_{now}, \infty, -r_{min}(t'_{now} - t_{now}))$
      **if** $n.left =$ **nil** $\wedge\, n.value = 0 \wedge n.offset = 0$ **then**
         **return nil**
      **else**
         **return** $n$
      **end if**
   **end if**
```

Algorithm 9 Shift the tree to represent only nodes from t'_{now} onwards; for use on the tree root only.

```
shiftTimeRoot$(n, t_{now}, r_{min}, t'_{now})$
   $n' \leftarrow$ shiftTime$(n, t_{now}, r_{min}, t'_{now})$
   **if** $n =$ **nil** $\vee\, n.left.valid$ **then**
      **return** $n$
   **else**
      **return** incrementOffset$(n.right, n.offset)$
   **end if**
```

Algorithm 10 Returns at what time in the future we will be able to accept a put of size x and TTL ℓ.

```
nextAccept($n, t_{now}, r_{min}, C, x, \ell$)
  $n' \leftarrow$ addPut($n, t_{now}, r_{min}, \ell, x$)
  **if** $n.offset + n.value \leq C$ **then**
    **return** $t_{now}$
  **else**
    $h \leftarrow t_{now} + (x-1)/r_{min} + 1$
    $l \leftarrow t_{now}$
    **while** $h - l > 1$ **do**
      $t \leftarrow l + (h-l)/2$
      $n' \leftarrow$ addPut(shiftTimeRoot($n, t_{now}, r_{min}, t$), $t, r_{min}, \ell, x$)
      **if** $n.offset + n.value \leq C$ **then**
        $h \leftarrow t$
      **else**
        $l \leftarrow t$
      **end if**
    **end while**
    **return** $h$
  **end if**
```

Bibliography

[ABKM01] David Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[B$^+$04] A. Bavier et al. Operating system support for planetary-scale network services. In *Proceedings of the USENIX Symposium on Design and Implementation (NSDI)*, March 2004.

[bit] Bittorrent goes trackerless: Publishing with bittorrent gets easier! `http://www.bittorrent.com/trackerless.html`.

[BMP03] Micah Beck, Terry Moore, and James S. Plank. An end-to-end approach to globally scalable programmable networking. In *FDNA*, 2003.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[BR03] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*, 2003.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[BSV03] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2003.

[BSW05] Hari Balakrishnan, Scott Shenker, and Michael Walfish. Peering peer-to-peer providers. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2005.

[BTC+04] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoff M. Voelker. Total Recall: System support for automated availability management. In *Proceedings of the USENIX Symposium on Design and Implementation (NSDI)*, 2004.

[Cat03] Josh Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[CCR03a] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Microsoft, 2003.

[CCR03b] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, December 2003.

[CDK+03a] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in a cooperative environment. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[CDK+03b] Russ Cox, Frank Dabek, Frans Kaahoek, Jinyang Li, and Robert Morris. Practical, distributed network coordinates. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2003.

[CDKR02] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks. In *Proceedings of the ACM SIGOPS European Workshop*, September 2002.

[CJK+03] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Anthony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, April 2003.

[CLL02] Jacky Chu, Kevin Labonte, and Brian Neil Levine. Availability and locality measurements of peer-to-peer file systems. In *Proc. of ITCom: Scalability and Traffic Control in IP Networks*, July 2002.

[CMM02] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving DNS using a peer-to-peer lookup service. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[CRL03] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3), August 2003.

[CRR$^+$05] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered dht applications. In *Proceedings of ACM SIGCOMM*, August 2005.

[DC99] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[DGH$^+$87] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 1987.

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

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proceedings of ACM SIGCOMM*, 1989.

[DLS$^+$04] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the USENIX Symposium on Design and Implementation (NSDI)*, 2004.

[Dou02] John Douceur. The Sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[DR01] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[DS] Frank Dabek and Emil Sit. Personal communication.

[DZD$^+$03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common API for structured P2P overlays. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[edo] eDonkey2000 – Overnet. `http://www.edonkey2000.com/`.

[FFM04] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the USENIX Symposium on Design and Implementation (NSDI)*, March 2004.

[fre] Freepastry. `http://freepastry.rice.edu/`.

[fre05] Freepastry release notes. `http://freepastry.rice.edu/FreePastry/README-1.4.1.html`, May 2005.

[GBL$^+$03] Indranil Gupta, Kenneth Birman, Prakash Linga, Al Demers, and Robbert Van Renesse. Kelips: building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2003.

[GDS$^+$03] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[GGG$^+$03] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM*, August 2003.

[GLR04] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the USENIX Symposium on Design and Implementation (NSDI)*, 2004.

[GLS$^+$04] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load balancing in dynamic structured P2P systems. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2004.

[GMG$^+$02] Krishna P. Gummadi, Harsha V. Madhyastha, Steven D. Gribble, Henry M. Levy, and David Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proc. OSDI*, 2002.

[gnu] Gnutella. `http://www.gnutella.com/`.

[GS03] Steven Gerding and Jeremy Stribling. Examining the tradeoffs of structured overlays in a dynamic non-transitive network, 2003. Class project: `http://pdos.lcs.mit.edu/~strib/doc/networkingfall2003.pdf`.

[GVC96] P. Goyal, H.M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of ACM SIGCOMM*, August 1996.

[HHL$^+$03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

[HKRZ02] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[ine] Inet topology generator.
`http://topology.eecs.umich.edu/inet/`.

[JK88] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM*, 1988.

[JSBM01] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS performance and the effectiveness of caching. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2001.

[JT75] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Arpanet RFC 677, January 1975.

[KBC$^+$00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[KHFP03] Eddie Kohler, Mark Handley, Sally Floyd, and Jitendra Padhye. Datagram congestion control protocol (DCCP). `http://www.icir.org/kohler/dcp/draft-ietf-dccp-spec-04.txt`, June 2003.

[KK03] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal hash table. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[KKD01] David Kempe, Jon Kleinberg, and Alan Demers. Spatial gossip and resource location protocols. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, July 2001.

[KR04] David R. Karger and Matthias Ruhl. Diminished Chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[KRRS04] Brad Karp, Sylvia Ratnasamy, Sean Rhea, and Scott Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[LCH$^+$05] Boon Thau Loo, Tyson Condie, Joseph Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[LHSH04] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph Hellerstein. The case for a hybrid P2P search infrastructure. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[LMR02] Nancy Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in content addressable networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[LNBK02] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, July 2002.

[LSG+04] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[LSM+05] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2005.

[LSMK05] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the USENIX Symposium on Design and Implementation (NSDI)*, 2005.

[M+03] Alan Mislove et al. POST: a secure, resilient, cooperative messaging system. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*, 2003.

[MCM01] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[MCR03] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2003.

[MD88] Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. In *Proceedings of ACM SIGCOMM*, 1988.

[Mer88] R. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Proceedings of the Annual International Cryptology Conference (CRYPTO)*, pages 369–378. Springer-Verlag, 1988.

[MGM05] Athicha Muthitacharoen, Seth Gilbert, and Robert Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT LCS, June 2005.

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

[MM02] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[MMGC02] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[MNJH04] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host identity protocol (work in progress). IETF Internet Draft, 2004.

[NL97] Jason Nieh and Monica S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[Pax97] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, 1997.

[Pit87] Boris Pittel. On spreading a rumor. *SIAM J. Applied Math*, 47, 1987.

[PRR97] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[PST$^+$97] Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Alan Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[Rab81] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *IFIP/ACM Middleware*, November 2001.

[REG$^+$03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, March 2003.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, August 2001.

[RGK$^+$05] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of ACM SIGCOMM*, August 2005.

[RGRK03] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. Technical Report UCB//CSD-03-1299, University of California, Berkeley, December 2003.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[RH03] Timothy Roscoe and Steven Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*, May 2003.

[Rhe03a] Sean Rhea. Epidemic algorithms at work. OceanStore Developers Mailing List, April 2003. `https://oceanstore.cs.berkeley.edu/mailarchive/oceanstore.0304/0012.html`.

[Rhe03b] Sean Rhea. Re: Epidemic algorithms at work. OceanStore Developers Mailing List, April 2003. `https://oceanstore.cs.berkeley.edu/mailarchive/oceanstore.0304/0013.html`.

[RHKS01] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. *Lecture Notes in Computer Science*, 2233:14–29, 2001.

[RK04] Matthias Ruhl and David R. Karger. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[RKCD01] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *NGC2001*, 2001.

[RL03] Rodrigo Rodrigues and Barbara Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT CSAIL, December 2003.

[RLB03] Sean Rhea, Kevin Liang, and Eric Brewer. Value-based web caching. In *Proceedings of the International World Wide Web Conference (WWW)*, May 2003.

[ron] Ron latency data. `http://nms.csail.mit.edu/ron/data/`.

[RRHS04] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph Hellerstein, and Scott Shenker. Brief announcement: Prefix hash tree (extended abstract). In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[RS04a] Venugopalan Ramasubramanian and Emin Gun Sirer. The design and implementation of a next generation name service for the Internet. In *Proceedings of ACM SIGCOMM*, August 2004.

[RS04b] Venugopalan Ramasubramanian and Emin Gn Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the USENIX Symposium on Design and Implementation (NSDI)*, 2004.

[rss] RSS protocol. Wikipedia. `http://en.wikipedia.org/wiki/RSS_(protocol)`.

[SAZ$^+$02] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*, August 2002.

[SCL$^+$05] Jeremy Stribling, Isaac G. Councill, Jinyang Li, M. Frans Kaashoek, David R. Karger, Robert Morris, and Scott Shenker. Overcite: A cooperative digital research library. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[SDR04] Emil Sit, Frank Dabek, and James Robertson. UsenetDHT: A low overhead Usenet server. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[SGG02] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, January 2002.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*, August 2001.

[SMPD05] Daniel Sandler, Alan Mislove, Ansley Post, and Peter Druschel. FeedTree: Sharing web micronews with peer-to-peer event notification. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2005.

[Str] Jeremy Stribling. Planetlab all-pairs ping. `http://www.pdos.lcs.mit.edu/~strib/pl_app/APP_README.txt`.

[STZ04] Subhash Suri, Csaba Toth, and Yunhong Zhou. Uncoordinated load balancing and congestion games in P2P systems. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[SW00] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, 2000.

[SW02] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, November 2002.

[VvRB02] Werner Vogels, Robbert van Renesse, and Ken Birman. The power of epidemics: Robust communication for large-scale distributed systems. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, October 2002.

[VYW$^+$02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[WBS04] Michael Walfish, Hari Balakrishnan, and Scott Shenker. Untangling the Web from DNS. In *Proceedings of the USENIX Symposium on Design and Implementation (NSDI)*, March 2004.

[ZDH$^+$02] Ben Y. Zhao, Yitao. Duan, Ling Huang, Anthony D. Joseph, and John D. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[ZHS$^+$04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[ZZJ$^+$01] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 2001.