Storing and Managing Data in a Distributed Hash Table

by

Emil Sit

S.B., Computer Science (1999); S.B., Mathematics (1999);
M.Eng., Computer Science (2000)
Massachusetts Institute of Technology

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

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 1, 2008

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

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

Accepted by . . . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

Storing and Managing Data in a Distributed Hash Table

by

Emil Sit

Submitted to the Department of Electrical Engineering and Computer Science
on May 1, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Distributed hash tables (DHTs) have been proposed as a generic, robust storage infrastructure for simplifying the construction of large-scale, wide-area applications. For example, UsenetDHT is a new design for Usenet News developed in this thesis that uses a DHT to cooperatively deliver Usenet articles: the DHT allows a set of N hosts to share storage of Usenet articles, reducing their combined storage requirements by a factor of $O(N)$. Usenet generates a continuous stream of writes that exceeds 1 Tbyte/day in volume, comprising over ten million writes. Supporting this and the associated read workload requires a DHT engineered for durability and efficiency.

Recovering from network and machine failures efficiently poses a challenge for DHT replication maintenance algorithms that provide durability. To avoid losing the last replica, replica maintenance must create additional replicas when failures are detected. However, creating replicas after every failure stresses network and storage resources unnecessarily. Tracking the location of every replica of every object would allow a replica maintenance algorithm to create replicas only when necessary, but when storing terabytes of data, such tracking is difficult to perform accurately and efficiently.

This thesis describes a new algorithm, Passing Tone, that maintains durability efficiently, in a completely decentralized manner, despite transient and permanent failures. Passing Tone nodes make replication decisions with just basic DHT routing state, without maintaining state about the number or location of extant replicas and without responding to every transient failure with a new replica. Passing Tone is implemented in a revised version of DHash, optimized for both disk and network performance. A sample 12 node deployment of Passing Tone and UsenetDHT supports a partial Usenet feed of 2.5 Mbyte/s (processing over 80 Tbyte of data per year), while providing 30 Mbyte/s of read throughput, limited currently by disk seeks. This deployment is the first public DHT to store terabytes of data. These results indicate that DHT-based designs can successfully simplify the construction of large-scale, wide-area systems.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

Previously Published Material

Portions of this thesis are versions of material that were originally published in the following publications:

> SIT, E., DABEK, F., AND ROBERTSON, J. UsenetDHT: A low overhead Usenet server. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems* (Feb. 2004).
>
> CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *Proc. of the 3rd Symposium on Networked Systems Design and Implementation* (May 2006).
>
> SIT, E., MORRIS, R., AND KAASHOEK, M. F. UsenetDHT: A low overhead design for Usenet. In *Proc. of the 5th Symposium on Networked Systems Design and Implementation* (Apr. 2008).

Acknowledgments

This thesis would not have been possible without collaboration with my colleagues, and the support of my friends and family.

My advisors, M. Frans Kaashoek and Robert Morris, have been an endless source of wisdom, optimism and advice; by example, they have taught me how to conduct and present systems research. In addition to their support and feedback, they have created a fun and engaging environment at PDOS.

The members of PDOS and other CSAIL systems students and faculty have been invaluable in preparing talks, thus in-directly refining the presentation of ideas in this thesis. In addition to my advisors, particular thanks go to Russ Cox, who can always be counted on to attend yet another practice and offer suggestions for improvement.

Systems research requires the development of significant software. Frank Dabek worked closely with me to develop many of the ideas in this thesis and implement the core Chord and DHash infrastructure. Josh Cates wrote the initial Merkle tree implementation and devised the earliest maintenance protocols used in DHash. Jeremy Stribling has also been a close collaborator on papers and code; he implemented the initial prototypes of Passing Tone and on-disk Merkle tree storage. Andreas Haeberlen implemented the early version of the simulator used to evaluate Passing Tone and Carbonite; his ability to transform ideas or questions into software and graphs overnight is inspiring. Max Krohn and Russ Cox were often sounding boards for my questions about coding and debugging; I am particularly grateful to them for taking on the burden of keeping the computer systems at PDOS running smoothly. Jinyang Li, Athicha Muthitacharoen, James Robertson and Benjie Chen also contributed code and ideas to what has become the work in this thesis.

Garrett Wollman provided the live Usenet feed from CSAIL used for evaluation and graciously answered many questions. Trace data used in Chapter 3 was provided by Vivek Pai and Aaron Klingaman. The evaluation in Chapter 6 made use of the PlanetLab and the RON test-beds. David Andersen helped solve many problems with accessing and using RON nodes. Additional nodes were provided by Magda Balazinska (University of Washington), Kevin Fu (University of Massachusetts, Amherst), Jinyang Li (New York University), and Alex Snoeren (University of California, San Diego).

My graduate studies were funded by the Cambridge-MIT Institute (as part of the Design and Implementation of Third Generation Peer-to-Peer Systems project), an NSF ITR grant (ANI-0225660 for Robust Large-Scale Distributed Systems), and various teaching assistant positions.

Graduate school is only partly about research. My life has been immeasurably enriched by my wife, Aleksandra Mozdzanowska, who has loved and supported me through my years in graduate school, and the many hobbies and distractions I acquired along the way. These last few years were made wonderful by the time I spent with my teachers and friends at the Back Bay and South Boston Yoga studios, most notably David Vendetti, whose wisdom has guided me and so many others. Finally, my parents deserve thanks for instilling a life-long interest in learning and supporting me along the way.

Contents

Chapter 1

Introduction

The Internet has transformed communication for distributed applications: each new system need not implement its own network, but can simply assume a shared global communication infrastructure. A similar transformation might be possible for storage, allowing distributed applications to assume a shared global storage infrastructure. Such an infrastructure would have to name and find data, assure durability and availability, balance load across available servers, and move data with high throughput and low latency. An infrastructure achieving these goals could simplify the construction of distributed applications that require global storage by freeing them from the problem of managing wide-area resources.

Distributed hash tables (DHTs) are a promising path towards a global storage infrastructure. Distributed hash tables are decentralized systems that aim to use the aggregate bandwidth and storage capacity from many distributed nodes to provide a simple storage interface: like regular hash tables, applications can *put* and *get* data from the DHT without worrying about how that data is stored and located. In addition to being easy for developers to use, DHTs are attractive to administrators because DHTs are self-managed: DHT software running on the individual nodes detect changes in the system and adapt accordingly without input from human operators. The attraction of such an infrastructure has led to DHT prototypes that have been used to develop a variety of distributed applications, such as wide-area file and content publishing systems [16, 53, 65, 69]. Commercial services such as Amazon.com's public Simple Storage Service (S3) and their internal DHT-inspired back-end, Dynamo [19], have demonstrated the utility of the *put*/*get* interface for building applications from shopping carts to expandable disks to photo-sharing services.

A distributed hash table must address the problems that it seeks to hide from developers and administrators. DHT algorithms must make intelligent decisions and manage resources to provide high-performance, scalable, and durable storage, without any centralized coordination. This task is not simple. Node populations are diverse, with storage and bandwidth resources that may vary widely across nodes. This complicates load balance and providing good routing and storage performance. Failures are a constant presence in large node populations [25]; a DHT must recover from failures to avoid losing data.

While early systems have been successful at improving routing performance and providing basic storage, no major applications have been deployed that use DHTs to durably store data

in the wide-area. Rather, these applications cache data stored elsewhere (e.g., the Coral CDN [21]), only support small data items (e.g., OpenDHT [17, 66]), or have successfully stored only limited amounts of data [86]. The critical requirement for storage applications is durable data storage: without durability, it is impossible to store hard-state in a DHT. Unfortunately, durability can be hard to achieve when failures are constantly affecting the active node population. Such failures affect availability and can also potentially result in loss of data.

This thesis focuses on storing and maintaining immutable data. Immutable data can be used to build interesting applications. Immutable data naturally lends itself to snapshot and archival applications such as the Venti archival backup system [60], the Cooperative File System [16], and the `git` distributed version control system [34]. It also forms the basis for mutable abstractions such as Ivy [53] and the Secure Untrusted Data Repository (SUNDR) [45]. Thus, focusing on immutable data frees the DHT from worrying about problems of consistency and conflicts but still enables interesting applications to be built.

To keep the techniques in this thesis concrete and relevant, we motivate the development of high-performance, durable DHT storage by building a Usenet server called UsenetDHT. In Section 1.1, we introduce Usenet and the challenges that UsenetDHT presents for DHT storage. The specific goals UsenetDHT induces are laid out in Section 1.2. Section 1.3 details the contributions of the thesis and Section 1.4 lays out the structure of the remaining chapters.

1.1 Motivating application: UsenetDHT

Usenet is a distributed messaging service that connects thousands of sites world wide. UsenetDHT is a cooperative server for participating in the global Usenet. UsenetDHT uses a DHT to share storage among multiple servers; it aims to bring incrementally scalable storage capacity and performance to Usenet servers [72, 75]. Usenet is an excellent target application to motivate DHT development for several reasons: the objects written are immutable and easily stored in a DHT, it naturally generates a vast amount of data, and Usenet operators can potentially benefit financially from the use of a DHT.

Usenet is highly popular and continues to grow: one Usenet provider sees upwards of 40,000 readers reading at an aggregate 20 Gbit/s [83]. Several properties contribute to Usenet's popularity. Because Usenet's design [4, 37] aims to replicate all articles to all interested servers, any Usenet user can publish highly popular content without the need to personally provide a server and bandwidth. Usenet's maturity also means that advanced user interfaces exist, optimized for reading threaded discussions or streamlining bulk downloads. However, providing Usenet service can be expensive: users post over 1 Tbyte/day of new content that must be replicated and stored.

The idea of full replication in the design of Usenet pre-dates modern networks. Full replication results in a tremendous bandwidth and storage cost: to just accept a full feed of Usenet would require at least a 100 Mbps link since users post over 1 Tbyte of data per day, corresponding to over 4 million articles. To pay for this capacity, Usenet providers charge users a fee—the provider promise to push replicas of any content posted by the user

into the overlay so that others may view it; symmetrically, it receives replicas of all other postings so that the user can benefit from having a reliable mirror of popular content locally. Unlike a CDN, where the publisher pays for all readers, each reader pays for his own access. UsenetDHT demonstrates how a DHT could significantly reduce the costs to the provider which would hopefully result in reduced costs for users or improved service.

UsenetDHT is best suited for deployment at institutions that can cooperate to reduce costs of operating a Usenet server. UsenetDHT uses a distributed hash table to store Usenet articles across participating servers. The servers arrange their external feeds so that they receive only one copy of each article, and store that one copy into the common DHT using the content hash of the article as the lookup key. UsenetDHT then floods information about the existence of each article (including its key) among all UsenetDHT servers. Instead of transmitting and storing N copies of an article (one per server), UsenetDHT requires only two for the entire deployment (for durability; more may be kept for performance). Thus, per server, UsenetDHT reduces the load of receiving and storing articles by a factor of $O(N)$.

The other bandwidth cost comes from supporting readers. UsenetDHT relies on caching at individual servers to ensure that articles do not get transmitted over the wide-area to a given server more than once. Thus, even if clients read all articles, UsenetDHT is unlikely to create more copies than Usenet. The remaining bandwidth used is a function of the percentage of articles that are read in the entire system. Statistics from MIT's news servers indicate that clients read only a small fraction of the articles their local server receives: clients at MIT read about 1% of the feed that MIT receives and forwards to other servers [68, 88]. This means that significant savings can be had if MIT were to partner with similar institutions to run UsenetDHT: the costs of supporting reads will be much smaller than the cost of fully replicating. Similarly, the pooled storage capacity would allow for a system with more total capacity than any single site may be willing to provide.

In addition to delivering cost savings, UsenetDHT is incrementally deployable and scalable. UsenetDHT preserves the NNTP client interface, allowing it to be deployed transparently without requiring that clients change their software. The use of an underlying DHT allows additional storage and front-end servers to be easily added, with the DHT handling the problem of re-balancing load. Additionally, UsenetDHT preserves the existing economic model of Usenet. Compared to alternative content distribution methods such as CDNs or `ftp` mirror servers that put the burden of work and cost on the publisher, UsenetDHT leaves the burden of providing service at the Usenet server. This means that the cost of running the system is still placed on the consumers of information, not the producers.

1.2 Goals

To support a system like UsenetDHT, we must ensure that the underlying DHT provides basic properties like node membership management, load balance, and incremental scalability. In terms of routing state, existing algorithms such as Chord [78], Accordion [46] and Y0 [27] provide low-latency, load-balanced routing. However, a DHT must also provide two important *storage* properties: efficient durability and performance.

1.2.1 Bandwidth-efficient durability

Most existing work in the distributed hash table community has been focused on achieving and understanding block *availability* [6, 7]. The availability of a block stored in a DHT can be measured as the percentage of time that the system is able to return the block when asked, out of the total amount of time that the system is operating. In distributed systems, it is hard to achieve availability because there are many sources of transient failure that the system can not predict or control [1]. For example, if a client's ISP's BGP configuration is incorrect and causes a network partition, the client will not be able to retrieve data even if the rest of the DHT is operating normally. Failures inside the network can be hard to diagnose as well. However, many systems do not truly require high availability—they are interested in having data *durability*. For typical Internet services, users can often tolerate temporary unavailability but not permanent loss. For example, UsenetDHT users are likely to tolerate temporary inaccessibility as long as they eventually get to read all the articles.

Durable data storage ensures that each piece of data is physically stored in the DHT for as long as the application requires. Note that for data to be highly available, it must typically be stored in a durable manner as well. Thus, durability can be thought of as a prerequisite for availability. Durability is necessary if DHTs are to be used to store hard-state. Without durable data, the safest option is to focus on storing soft state: for example, cached data (*e.g.* Coral [21]) or data that is refreshed periodically by the client (*e.g.* OpenDHT [41]).

Redundancy is used to provide both data availability and durability. As redundancy is added, the system becomes more resilient to failures. DHT algorithms must decide how much redundant data to create and which nodes should store it. The system may also need to reclaim space periodically to ensure adequate capacity. These decisions must take into account the constrained resources (bandwidth and storage) of each node.

Further, because nodes are subject to failure, the DHT must monitor the availability of data and re-create lost redundant data when necessary. Indeed, it has been argued that DHTs are unsuitable for bulk data storage because they use too much bandwidth to maintain redundant data for availability in the face of nodes leaving and joining the system [7].

The two key threats to durability are disk failure and permanent departures. Both events result in loss of data. Other failures, such as power failures, software bugs, and network failures, are transient and only affect availability. Unfortunately, it is difficult in practice to distinguish a permanent failure from a transient failure. In a large network without central coordination, nodes (and hence the data they hold) are monitored remotely. For example, `a.mit.edu` might monitor `b.berkeley.edu`; if `a` detects that `b` is not reachable, it can not know for certain if `b` will ever return with its data intact.

A naïve scheme for providing durability could use significant bandwidth if it eagerly repairs lost redundancy each time a node goes offline and aggressively reclaims excess redundancy. Earlier work in DHash proposed essentially this scheme [12] while trying to achieve availability: erasure coded fragments were spread across a replica set that was contiguous in the overlay's structure. Every time a node joined or left that portion of the overlay, the replica set changed, requiring the movement (i.e., deletion) or creation of a fragment. This resulted in a continuous stream of repair traffic. One alternative might be

to provide more redundancy than actually desired and then only repair lazily, as done by the Total Recall system [6]. This could potentially be wasteful if the excess redundancy is too high, or insufficient if excess redundancy is too low—Total Recall's efficiency depends on its ability to correctly estimate the node failure rate. Total Recall also causes bursts of maintenance traffic when block redundancy falls below the desired threshold.

High-bandwidth maintenance would undermine the benefits of UsenetDHT's design: in Usenet, while expensive, total replication's cost is bounded by the cost of copying every article to each participating host once. After the copy is done, no additional bandwidth cost is incurred, regardless of whether any particular host fails. However, if UsenetDHT were run on a DHT with an inefficient data maintenance scheme, each node failure would reduce availability of blocks, and would result in some repair action. These repair actions would result in an essentially unbounded amount of traffic since nodes failures will always occur. Thus, to effectively support applications like UsenetDHT, efficient durability maintenance is required.

1.2.2 Performance

The other major challenge in building DHTs is achieving performance: low latency and good throughput. This requires both adequate network performance and disk performance. To get good network performance, DHT routing algorithms must accommodate nodes spread across the globe, with access link capacities that vary by orders of magnitude, and experiencing varying degrees of congestion and packet loss. Routing latency has largely been minimized in prior work [17, 46], but such systems have not focused on achieving high disk performance.

High disk throughput is normally achieved by minimizing disk seeks and ensuring that data is read or written sequentially from the platter. However, DHT applications by nature face a highly randomized read and write workload that may make sequential I/O hard to achieve; for example, there is little guarantee that two Usenet article writes to the same newsgroup will be stored on the same machine, even if they are part of the same larger binary posting. This means that reads will necessarily incur seeks. In addition, a DHT also has two major background tasks that may cause additional seeks: object expiration and replica maintenance.

Object expiration is of particular importance for the UsenetDHT application. The volume of continuous writes implies that it is not possible to keep Usenet articles forever: the system must make a decision as to how long to *retain* articles based on available disk capacity. A natural storage implementation may make the expiration of any individual object expensive in terms of seeks—unlinking a file or updating a database both require several seeks.

The design of replica maintenance, where a DHT replaces lost replicas to prevent data loss or unavailability, also affects performance. The goal of replica maintenance is to avoid losing the last replica, but without making replicas unnecessarily, since objects are expensive to copy across the network. To achieve this, a maintenance algorithm must have an accurate estimate of the number of replicas of each object in order to avoid losing the last one. Since DHTs partition responsibility for maintenance across nodes, a simple solution would have each DHT node check periodically with other nodes to determine how many replicas exist for objects in its partition. However, naïvely exchanging lists of object identifiers in order to see

which replicas are where can quickly become expensive [33].

A more efficient mechanism would be to simply exchange lists once and then track updates: a synchronization protocol (e.g., [12, 51]) can identify new objects that were added since the last exchange as well as objects that have been deleted to reclaim space. However, this alternative means that each node must remember the set of replicas held on each remote node that it synchronizes with. Further, each synchronization must cover objects inserted in well-defined periods so that when a node traverses its local view of replicas periodically to make repair decisions, it considers the same set of objects across all nodes. These problems, while solvable, add complexity to the system. The storage and traversal of such sets can also cause disk activity or memory pressure, interfering with regular operations.

To successfully and scalably deploy UsenetDHT, the implementation of the DHT must use algorithms and data structures that minimize the number of disk operations required to support both regular operations and background maintenance tasks.

1.3 Contributions

This thesis shows how to structure a distributed hash table to provide high performance, durable data storage for large, data-intensive applications. The techniques to achieve this are explained in terms of the MIT Chord/DHash DHT [14, 16, 17]. A complete implementation of these techniques can run the RON test-bed [2] and support UsenetDHT [72]: while a complete Usenet feed is estimated at over 1 Tbyte per day, DHash is able to support the highest capacity feed we have access to, which inserts at 175 Gbyte per day.

To handle maintenance under the workload of Usenet, we introduce *Passing Tone*. Passing Tone is ideal for maintenance of objects in a system with constant writes resulting in large numbers of objects and where objects are also continuously expired. For such workloads, Passing Tone enables distributed hash tables such as DHash to maintain data efficiently in a network of distributed hosts, despite permanent data loss due to disk failures and transient data loss due to network outages.

Passing Tone's design is driven by the goal of providing durability by keeping a threshold number of replicas while minimizing the number of coordinating nodes. This latter requirement keeps the implementation efficient (and simple): in the normal case, a Passing Tone node communicates with only two other nodes and requires no additional maintenance state beyond the objects it stores locally.

For a given object, instead of having the object's successor ensure that sufficient replicas exist, Passing Tone has all nodes in the object's successor list ensure that they replicate the object. Each Passing Tone node makes maintenance decisions by synchronizing alternately with its successor and predecessor against the objects it stores locally. Synchronization compares the list of objects held remotely against those stored locally and identifies objects that the node should replicate but doesn't. These objects are then pulled from its neighbors. In this way, Passing Tone ensures that objects are replicated on the successor list without having to explicitly track/count replicas, and without having to consider a consistent set of objects across several nodes.

By taking advantage of redundancy that is produced during transient failures, Passing Tone rarely has to produce new replicas. In fact, Passing Tone need not explicitly distinguish between permanent and transient failures but still provides efficiency. In environments of cooperating nodes whose primary data loss is due to disk failure, this thesis shows that Passing Tone solves the problem of providing durability in a DHT while efficiently managing bandwidth and storage.

This thesis demonstrates a complete implementation of Passing Tone. The implementation of Passing Tone in DHash lays out data and metadata structures on disk efficiently to make new writes, expiration and maintenance efficient. Overall performance is provided using a number of techniques, ranging from recursive lookup routing to careful engineering of the implementation to avoid or batch expensive disk I/O operations.

More concretely, the contributions of this thesis are:

- Passing Tone, a distributed algorithm for maintaining durability in the face of transient failures;
- An implementation of Passing Tone in the DHash DHT, and an evaluation of the durability provided and bandwidth used by Passing Tone on the RON test-bed; and
- The design, implementation and evaluation of the UsenetDHT server in a wide-area deployment.

The resulting implementation of DHash scales linearly in read and write throughput as nodes are added; on a 12-node wide-area network, UsenetDHT performance is limited by disk seeks but supports client reads at 30 Mbyte/s and accepts writes at over 2.5 Mbyte/s. Microbenchmarks show that DHash with Passing Tone scales linearly to provide additional performance as machines are added. Over a year-long trace of failures in the distributed PlanetLab test-bed (a deployment environment hosted at universities, similar to our targeted deployments, though with limited resources), simulations show that Passing Tone and DHash provide durability efficiently despite frequent transient failures and many permanent failures.

1.4 Summary

Distributed hash tables organize wide-area resources into a single logical storage system. By handling the problems of storing data in the wide-area, distributed hash tables may greatly simplify the development of distributed applications. DHT algorithms must be carefully designed so that background tasks such as data maintenance do not affect the ability of individual nodes to perform regular operations. Chapter 2 introduces the workload of Usenet and describes the major challenges involved in building a high-performance DHT in detail.

To show how to address these challenges, Chapter 3 presents Passing Tone, describing in detail the fundamentals that underlie our maintenance approach and the other design points that must be considered. Achieving correctness and adequate performance also requires a significant amount of implementation work; this work is summarized in Chapter 4.

Chapter 5 demonstrates the utility of the DHT approach a by presenting of the design of UsenetDHT. We evaluate the implementation of effectiveness of Passing Tone and UsenetDHT in Chapter 6, present related work in Chapter 7 and conclude in Chapter 8.

Chapter 2

Requirements and challenges

Distributed hash tables must solve numerous problems in order to successfully store bulk data. For example, they must deal with problems of both node management as well as data management. The solution to these two classes are logically separate and invisible to applications—the logical structure of DHT-based applications is shown in Figure 2-1. The application accesses the DHT using a simple key-value storage interface; this interface hides the complexity of the actual DHT implementation.

The DHT implementation contains the logic for managing the constituent nodes, and then storing and maintaining the data. Nodes are organized by a routing layer that maintains routing tables and adapts them dynamically as nodes join and leave. The storage layer uses individual nodes as hash table buckets, storing a fraction of the application's data on each one. Data maintenance, in the face of failures, is handled in the storage layer as well.

The problem of node management and routing is well-studied. We rely on existing protocols such as Chord [17, 77] and Accordion [46] to handle this problem. This chapter outlines the challenges faced by DHash in providing durable data storage and the problems that Passing Tone must solve. We begin with a description of the requirements induced by UsenetDHT and other applications with high request rate and high data volume.

2.1 Workloads and requirements

This thesis focuses exclusively on providing support for *immutable data* in the form of content-hashed objects. The name comes from the fact that the keys used for these objects are derived from a cryptographic hash of the object's content. In particular, our implementation sets the key for such objects to be the SHA-1 hash of the content. This allows data-integrity to be checked by all nodes in the system. Such objects are ideal for Usenet servers—the contents of a Usenet article are naturally immutable and, with content-hash checks, cannot be corrupted without detection by the system.

DHash must achieve both durability and performance in order to support highly demanding applications. Usenet servers, for example, must scale to support potentially thousands of simultaneous reads and writes. To understand more precisely the requirements upon DHash,

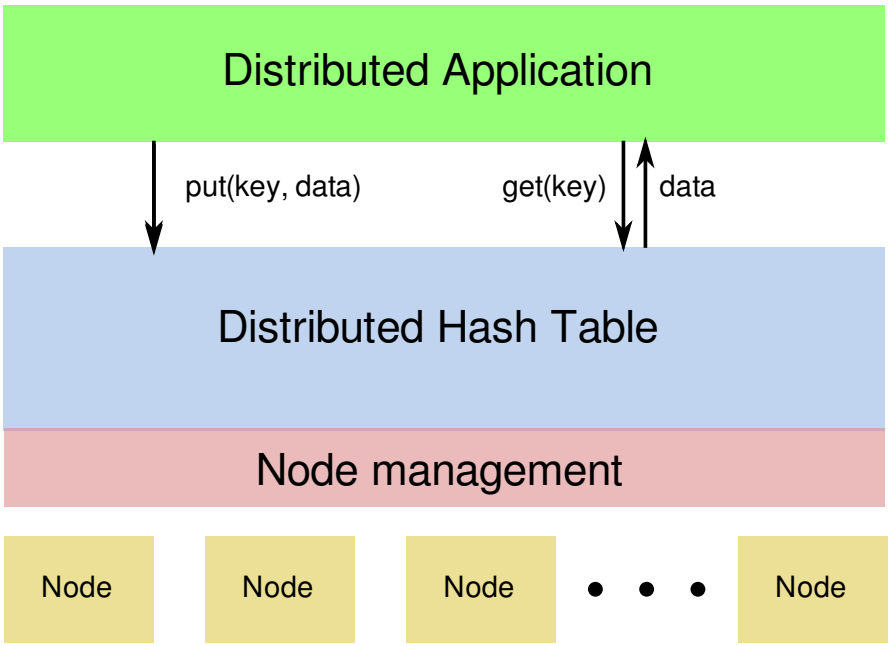


Figure 2-1: Logical structure of DHT-based applications. The DHT and its node management layer handles all issues related to managing a changing population of nodes and laying out data across those nodes. The distributed application itself uses a simple key-value storage interface to add and retrieve data from the system.

we will examine the needs of Usenet servers in detail.

The total volume exchanged over Usenet is hard to measure as each site sees a different amount of traffic, based on its peering arrangements. An estimate from 1993 showed an annual 67% growth in traffic [81]. Currently, in order to ensure avoid missing articles, top servers have multiple overlapping feeds, receiving up to 3 Tbyte of traffic per day from peers, of which approximately 1.5 Tbyte is new content [23, 26].

A major differentiating factor between Usenet providers is the degree of article *retention*. Because Usenet users are constantly generating new data, it is necessary to *expire* old articles in order to make room for new ones. Retention is a function of disk space and indicates the number of days (typically) that articles are kept before being expired. The ability to scale and provide high performance storage is thus a competitive advantage for Usenet providers as high retention allows them to offer the most content to their users. For example, at the time of this writing, the longest retention available is 200 days, requiring at least 300 Tbyte of data storage. Thus, a major requirement for DHash will be to provide *durable storage* with support for *expiration*. This storage must *scale easily* so that additional disks and servers can be added as needed to provide additional retention.

Usenet traffic consists of both small text-only articles and multi-media files that are transmitted as large, binary-encoded articles. The volume of text articles has remained

relatively stable for the past few years at approximately 1 Gbyte of new text data daily, from approximately 400,000 articles [28]. Growth in Usenet has largely been driven by increased postings of binary articles. As a result of the growth in traffic, being a full Usenet provider has become increasingly demanding. Each server that wishes to carry the full content of Usenet (a "full feed") must receive and store more data than a 100 Mbit/s connection can support. Top providers such as `usenetserver.com` and GigaNews propagate articles at dedicated peering points and have large data centers for storing (and serving) Usenet articles . To match these providers, DHash must support good *write throughput*.

Similarly, the read workload at major news servers can be extremely high: on a weekend in September 2007, the popular `usenetserver.com` served an average of over 40,000 concurrent connections with an average outbound bandwidth of 20 Gbit/s [83]. This suggests that clients are downloading continuously at an average 520 Kbit/s, most likely streaming from binary newsgroups. DHash must also provide high *read throughput*.

With such workloads, it is an expensive proposition for a site to provide many days of articles for readers. Usenet's economics allow providers to handle the high costs associated with receiving, storing, and serving articles. Perhaps surprising in the age of peer-to-peer file sharing, Usenet customers are willing to pay for reliable and high-performance access to the content on Usenet. Large providers are able to charge customers in order to cover their costs and produce a profit. In practice, most sites do not carry a full feed and instead ask their upstream providers to forward some subset of the total articles. For example, universities and other smaller institutions may find it difficult to bear the cost of operating an entire Usenet feed. UsenetDHT is an approach to bring these costs down and allow more sites to operate Usenet servers.

2.2 DHT challenges

Despite the fact that Usenet only requires immutable replicated storage, it is still difficult to provide a DHT that can serve Usenet effectively. DHash must enable UsenetDHT to accept writes as fast as its incoming feeds, to durably retain data for as long as the server desires (e.g., up to 200 days), and to efficiently expire older articles after the retention period. While performing these functions, DHash must support high read performance.

Providing these properties is dependent on how and where DHash is deployed. We envision a deployment of DHash nodes for Usenet in a manner similar to the PlanetLab distributed test-bed [59]: individual sites, typically well-connected universities or companies, will contribute one or more machines to the system and allow all other participants to share access to the machine. Thus, there will be many wide-area links involved. While many of these links will be well-provisioned, there is still expense associated with using them. DHash should seek to minimize wide-area bandwidth usage when feasible.

2.2.1 Failure handling

In a distributed system, failures are a common occurrence, affecting both data durability and availability. To provide durability, a DHT must conduct maintenance to deal with these failures.

Failures fall into two classes: permanent and transient. Transient failures, where a node fails after a network or power outage, result only in loss of availability. Permanent failures can cause data loss, affecting durability. The possibility of transient failures complicates providing durability efficiently. Because bandwidth is expensive, we do not want to make new copies in response to transient failures: repairs in response to transient failures represent wasted effort with regards to durability.

Permanent failures themselves can be disk failures or permanent node departures. Permanent departure rates are not well studied, especially in systems with stable servers like Usenet (and UsenetDHT). Existing studies showing rapid permanent departures have largely focused on file-sharing systems like Overnet [5]. For the purposes of this thesis, we will assume that, in server applications, permanent departures reflect planned situations and can thus be handled with explicit repairs. Thus, we focus on the problem of dealing with disk failures and not permanent departures.

A system that could remotely distinguish permanent disk failures from transient failures would need only make repairs after permanent disk failures, which are rare relative to transient failures. Such a system however must still be designed and parametrized carefully. A basic DHT approach to durability might begin by using replication, making two copies of each object. Upon detecting that one of the two replicas had suffered a disk failure, the DHT could make a new replica to ensure that the failure of the other disk would not lead to data loss. However, the simple policy of maintaining two replicas will not be sufficient because failures may come in bursts, faster than the system can create additional replicas.

To understand this more carefully, it is useful to consider the arrival of permanent disk failures as having an average rate and a degree of burstiness. A system must be able to cope with both to provide high durability. In order to handle some average rate of failure, a high-durability system must have the ability to create new replicas of objects faster than replicas are destroyed. Whether the system can do so depends on the per-node network access link speed, the number of nodes (and hence access links) that help perform each repair, and the amount of data stored on each failed node. When a node n fails, the other nodes holding replicas of the objects stored on n must generate replacements: objects will remain durable if there is sufficient bandwidth available on average for the lost replicas to be re-created. For example, in a symmetric system each node must have sufficient bandwidth to copy the equivalent of all data it stores to other nodes during its lifetime.

If nodes are unable to keep pace with the average failure rate, no replication policy can prevent objects from being lost. These systems are *infeasible*. If the system is infeasible, it will eventually "adapt" to the failure rate by discarding objects until it becomes feasible to store the remaining amount of data. A system designer may not have control over access link speeds and the amount of data to be stored, though choice of object placement can improve the speed that a system can create new replicas.

A burst of failures may destroy all of an object's replicas if they occur before a new replica can be made; a subsequent lull in failures below the average rate will not help replace replicas if no replicas remain. For our purposes, these failures are *simultaneous*: they occur closer together in time than the time required to create new replicas of the data that was stored on the failed disk. Simultaneous failures pose a constraint tighter than just meeting the average failure rate: every object must have more replicas than the largest expected burst of failures. We study systems that maintain a target number of replicas in order to survive bursts of failure; we call this target r_L.

Higher values of r_L do *not* allow the system to survive a higher average failure rate. If failures were to arrive at fixed intervals, then either $r_L = 2$ would always be sufficient, or no amount of replication would ensure durability. If $r_L = 2$ is sufficient, there will always be time to create a new replica of the objects on the most recently failed disk before their remaining replicas fail. If creating new replicas takes longer than the average time between failures, no fixed replication level will make the system feasible; setting a replication level higher than two would only increase the number of bytes each node must copy in response to failures, which is already infeasible at $r_L = 2$. The difficulty in configuring a replication level is knowing the burst rate of permanent failures. Fortunately, most scenarios we might encounter are able to survive with $r_L = 2$: disk failures and other permanent failures are sufficiently rare, and disks are small enough, that we have enough bandwidth to duplicate disks before their failure.

To summarize, the key problems that a DHT must solve in providing durability are first, to find a way to ignore transient failures so that maintenance can focus on permanent failures, and second, to select a reasonable replication level for an application's deployment environment. While it is impossible to distinguish between disk failures and transient failures using only remote network measurements, techniques from Carbonite [13] allow a system to approximate this ability. We defer discussion on the second problem, leaving r_L to be a tunable parameter for our system and giving some rudimentary guidance for the system operator. Passing Tone does its best to maintain the r_L set by the operator.

2.2.2 Limited disk capacity

While modern machines are well-provisioned with disks, the constant write workload of Usenet can quickly fill any sized disk. A DHT must deal with this in two ways. First, it must ensure that storage load is balanced evenly and fairly across all participating nodes, so that all disks fill more-or-less simultaneously. Secondly, it must provide a mechanism for ensuring that objects can either be deleted or removed from the system to reclaim space.

Load balance. Proper load balance ensures that available resources are used uniformly. In addition to distributing work in the system, fair load balance allows the system the luxury of allowing a single disk full to approximate a system capacity limit. That is, if each node's disk fills at approximately the same rate, a single EDISKFULL (or EQUOTA) error can be treated as an indication that other disks are also approximately full and that the application should consider notifying the administrator that more capacity must be added.

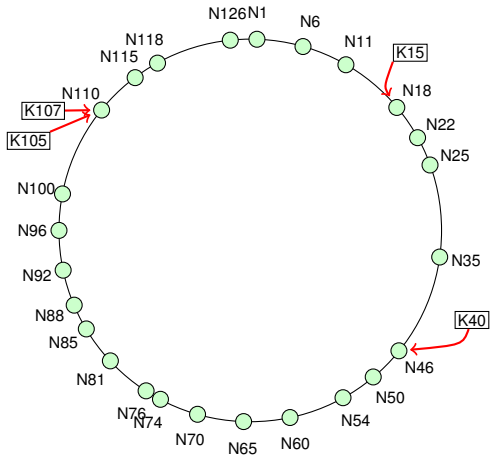


Figure 2-2: An example Chord ring with 7-bit identifiers. Nodes are assigned identifiers at random and arranged clockwise in increasing order. Keys are assigned to the first node clockwise; this node is called the *successor* of the key.

We use the Chord protocol to organize nodes and simultaneously provide load balance [16, 77]. Chord is a derivative of *consistent hashing* [38] that can handle a changing node population. As with consistent hashing, nodes are organized into a logical identifier space. Each node is given a unique identifier: Chord uses the space of 160-bit integers, corresponding to the size of a SHA-1 cryptographic hash. Nodes are organized in increasing numeric order, wrapping around after the highest identifier; this organization is shown in Figure 2-2. The basic relationship in Chord and other consistent hashing schemes is the *successor*. The successor of a key k is the first node clockwise of k. In Figure 2-2, the successor of key 15 is node 18. Chord assigns responsibility of objects to the successor of their key.

In a consistent hashing system, assuming a uniform read and write workload, the amount of load perceived by a given node is proportional to the fraction of the identifier space that lies between it and its predecessor. Thus, to achieve load balance, node identifiers must be assigned such that all nodes are spaced in the identifier space proportional to their capacity (or, in a homogeneous system, evenly). Consistent hashing and Chord's identifier assignment strategy provides a basic degree of randomized load balance. However, random node identifier assignment can still lead to a degree of imbalance; also, random identifier assignment does not take advantage of nodes with more capacity—for example, with more disk capacity or bandwidth.

Chord deals with this by way of *virtual nodes*: a single node joins the ring multiple times by choosing multiple identifiers. When Chord is used with DHash, each node uses a single database to store data across all of its virtual nodes. This ensures that Chord can adapt the number of virtual nodes in response to a change in relative capacity without forcing DHash re-balance the on-disk storage.

Different algorithms for choosing identifiers for virtual nodes have been proposed. Some

algorithms continue to randomly distribute virtual node identifiers through the entire identifier space but may choose them statically [16, 19] or dynamically adapt them as the membership changes [40]. To deal with heterogeneity in resources, the number of virtual nodes is varied, again either statically [14] or dynamically [27].

While the ideal scheme is dynamic and sets the number of virtual nodes so that the load is balanced and fair, often a static scheme suffices. The need for and benefit of a dynamic scheme depends on rate of change in the overall system population and capacity. A system that grows slowly, such as Amazon's Dynamo, may succeed with carefully managed static assignments [19]. A more dynamic system may require advanced techniques such as those used by Y0 [27].

A question for deployments is whether additional virtual node assignment techniques must be implemented to ensure adequate load balance. The deployment in this thesis does not anticipate rapid growth; thus our system has each node run a fixed number of virtual nodes, selected by the administrator, to balance the load across the system.

Reclaiming space. When the overall system disk capacity is reached, applications wishing to store new data must reclaim space by deleting existing data. Adding deletion to a DHT can be difficult because each delete operation needs to be propagated to nodes that are offline at the time of deletion. Such nodes may have a copy of deleted objects but be temporarily unavailable; it may come back online in minutes or months with data that had been deleted. At that time, the DHT maintenance algorithms may decide that the object's replication levels are dangerously low and begin to re-replicate it, effectively reversing the delete operation.

For UsenetDHT, expiration is a natural way to reclaim disk space, as it corresponds directly to how Usenet servers normally manage disk space. The DHash implementation extends *put* operation to take a key and an object and store it in the system until a *user-specified expiration* time. This allows the application to specify to the DHT how long the write must be durably retained. Up until that time, DHash guarantees that a *get* operation for the corresponding key will return the same object. By associating an expiration time with each object, assigned by the application, all nodes can independently agree on whether or not an object can be safely removed.

Deleting expired objects can be expensive: two obvious designs for data storage are storing objects in a single database or storing objects one per file. Both of these will require at least one seek per object to delete an object; this seek competes with regular read and write operations for the disk arm and can thus greatly reduce throughput and performance. A DHT should ensure that such interference is kept to a minimum.

Failure to account for expiration in the maintenance process can lead to repairs of expired objects. Such *spurious repairs* can occur if expirations are processed in bulk, a choice that makes sense if the overhead of expiring many objects approximates the cost of expiring just one. Unsynchronized bulk expirations could cause one node to expire an object before another. The maintenance process would then detect the loss of a replica on the former node and re-create it from the replica on the latter node. Repairing expired objects represent a waste of system resources since, by setting the expiration time, the application has indicated that a

given object no longer requires durability.

Thus, it is important that a DHT provide the ability to reclaim space but the design and implementation of the expiration process must be done with care to provide the proper semantics and avoid reducing overall performance.

2.2.3 Managing object movement

Load balance for data across nodes is important but it is also important that data be spread across nodes in a way that makes dealing with node population changes efficient. When a node fails, its contents must be re-created from other replicas. How many other nodes are involved in that re-creation? When a node joins, does data need to move in order to ensure that it can be efficiently located? This aspect of load balance is controlled by the choice of where replicas are placed; that is, how members of the replica set are selected.

When nodes choose the members of the replica set at random, the system is said to use random placement. Placement of replicas on random nodes means that node additions will never force object migration: random placement implies that the successor node must maintain a mapping from keys to the randomly selected nodes, in order to allow subsequent reads (and writes) to locate a replica. Random placement also spreads the objects that a given node monitors across all nodes—this can be help repair performance because it means that a single failure will result in repairs initiated by and involving all other nodes in the system, and thus complete quickly due to high parallelism.

However, random placement has several disadvantages as well. First, the indirection through the successor node means that read and write performance will require an extra round-trip for the client. This extra round-trip and the variance it introduces can hurt performance [19]. The importance of this mapping in supporting reads and writes means that the mapping itself must be replicated (as in Total Recall [6]), or at least, periodically re-created (similar to GFS [25]) and kept in sync across multiple nodes. Otherwise, the failure of the indirection node will lead to the inability to access data that is actually available in the system. Finally, the mapping must be maintained—as nodes fail, they must be removed from the entries for the objects they hold. Over time, each node will have to monitor every other node in the system. As the number of nodes in the system grows, this monitoring will become difficult to complete accurately and in a timely fashion due to slow or failed nodes. For these reasons, truly random placement is difficult to implement at scale in a wide-area environment with changing node population.

One alternative to true random placement is to limit the placement to be random within a small number of consecutive nodes: for example, r_L replicas can be placed among the nodes of the successor list. This avoids the problem of global monitoring but still allows the system to avoid migrating data on most node additions. However, unlike true random placement, some migration is unavoidable as the system grows since replicas outside of the successor list are not monitored and may be considered lost by a maintenance algorithm.

Finally, we can consider a placement policy that only uses the first r_L successors of a node. A successor placement policy limits where objects are stored so that they can be located easily and each node monitors only other nearby nodes. For example, DHash stores the

copies of all the objects with keys in a particular range on the successor nodes of that key range; the result is that those nodes store similar sets of objects, and can exchange summaries of the objects they store when they want to check that each object is replicated a sufficient number of times [12]. Glacier behaves similarly though instead of successive successors, it uses a deterministic calculation of secondary keys [33]. However objects may constantly migrate as the node population changes, using excessive bandwidth and potentially interfering with normal operation [19]. Passing Tone aims to avoid this problem while using successor placement.

2.2.4 Efficient maintenance

A maintenance algorithm must balance the desire to minimize bandwidth usage (e.g., by not repeatedly exchanging object identifier lists and not creating too many replicas) with the need to avoid storing and updating state about remote nodes. This problem largely revolves around deciding what information to keep on each node to make identifying objects that need repair easy and efficient.

A bandwidth-efficient maintenance algorithm would only generate repairs for those objects that have fewer than r_L remaining replicas. Such an algorithm would minimize replica movement and creation. A straightforward approach for achieving this might be to track the location of all available replicas, and repair when r_L or fewer remain. In a peer-to-peer system, where writes are sent to individual nodes without going through any central host, tracking replica locations would require that each node frequently synchronize with other nodes to learn about any new updates. The resulting replica counts would then periodically be checked to identify objects with too few replicas and initiate repairs.

The difficulty with this approach lies in storing state about replica locations in a format that is easy to update as objects are added and deleted but also easy to consult when making repair decisions. For ease of update and to allow efficient synchronization, information about replicas on each node needs to be kept in a per-node structure. However, to make repair decisions, each node requires a view of replicas over all nodes, to know how many replicas are currently available. While these views can be derived from one another, with millions of objects across dozens of nodes, it is expensive to construct them on the fly.

For example, Merkle trees can be used to reconcile differences between two nodes efficiently over the network [12]. However, constructing new Merkle trees per synchronization operation is CPU intensive; retaining multiple trees in memory would use significant memory. Storing both synchronization trees and aggregate counts is possible but would likely cause random disk I/O. Each of these scenarios would interfere with the node's ability to perform useful work: accepting new writes and servicing client reads.

Worse, even if state about replica locations can be efficiently stored, continuous writes and deletions means that any local state quickly becomes stale. Unfortunately, if information about objects on a remote node is stale—perhaps because an object was written to both the local and remote node but the local and remote have not yet synchronized—the local node may incorrectly decide to generate a new replica. Such spurious repairs can be bandwidth intensive and are hard to avoid.

The intermediate synchronization state must also be stored in memory or on disk. As the number of objects stored per node increases, a naïve implementation will place either extreme memory pressure or heavy seek traffic on the node.

The main challenge in the design of Passing Tone is avoiding unnecessary repairs (because making unnecessary copies is expensive), while at the same time avoiding disks seeks and being careful about memory usages. Meeting this challenge requires an efficient plan to synchronize nodes to identify objects that must be re-replicated.

2.2.5 I/O performance

DHash must accept and serve requests corresponding to UsenetDHT writes and reads. Achieving performance in servicing reads and writes is particularly important. This requires good network throughput and good disk performance. Network throughput requires that individual operations have low network overhead and can identify the relevant nodes quickly. When possible, the system should also offer a choice of nodes from which to satisfy reads so that the highest bandwidth path can be used.

Disk performance can be difficult because DHT workloads, and especially Usenet workloads, are randomly distributed: for example, successive reads at a single node are unlikely to result in sequential disk I/O. Writes can be optimized using an append-only log, but care must be taken to update any associated indices (needed for reads) efficiently. To ensure durability, it may be required to execute these writes synchronously, requiring on the order of one or two seeks per write. Due to the latency of typical disk seeks, this will limit each individual disk to 100 I/O operations per second. Since the resulting DHT workload will be seek heavy, a DHT must design and implement its storage system efficiently. to minimize the number of machines (and disks) needed to support application requests.

The difficulty in achieving performance is complicated by the need to support expiration: expiration involves deleting many individual objects both from disk and from the synchronization data structures. Again, a naïve design for object storage on disk can result in multiple seeks per expiration operation. Storing each object as a separate file, for example, would mean that expiration requires unlinking the file: on a standard Unix file system, unlinking the file can require several seeks in order to update the directory as well as the inode and block free lists.

Efficient use of local resources such as CPU and memory is also required during data maintenance. Excessive use of local CPU, disk or memory resources may interfere with normal operation of the system. For example, high CPU load may affect the ability to respond to even simple RPCs in a timely fashion; excessive disk usage reduces the ability to process reads and writes.

The design of a DHT must address the question of how best to layout storage, and how best to use other resources, to ensure that good I/O performance can be provided.

Chapter 3

Achieving efficient durability

To meet the challenges of providing durability in a DHT capable of supporting high-throughput read and write workloads, this chapter introduces the Passing Tone maintenance algorithm.* We present Passing Tone in the context of the DHash DHT, which stores immutable replicated objects. DHash stores object replicas initially on the first r_L successors of the object's key; this set of nodes is called the *replica set*. Passing Tone provides availability and durability by ensuring that all members of the replica set have a replica of the object, despite any changes in node population.

Passing Tone deals with the specific challenges of maintenance in systems with high performance demands by:

- keeping extra replicas to mask transient failures and provide durability;
- distributing work of replica maintenance for a given object across the replica set (instead of solely its successor), to provide efficiency;
- locating replicas to optimize read and repair parallelism; and
- efficiently storing object data and metadata to minimize seeks required for both writing new objects and expiring old ones.

Since no algorithm can truly guarantee durability (e.g., hostile alien terraforming of the earth may result in data loss), Passing Tone takes a best-effort approach to providing durability for a per-object, user-specified expiration time. Passing Tone shows that it is possible to achieve durability using a simple incremental repair algorithm, without worrying about distinguishing different types of failures.

The design of Passing Tone builds on prior and related work such as Antiquity [86], Carbonite [13], Glacier [33], and Total Recall [6]. Compared to these systems, Passing Tone provides a more minimal mechanism, aimed at avoiding inconsistencies that could lead to spurious repairs. Passing Tone's simplicity leads to an efficient and concise implementation (see Chapter 4).

*The name "Passing Tone" draws an analogy from the musical meaning of "Chord" to the action of the maintenance algorithm: passing tones are notes that pass between two notes in a chord.

3.1 Algorithm

Passing Tone aims to maintain at least r_L replicas per object. To do this, as a system, Passing Tone continuously monitors the number of reachable copies of each object. If the number of reachable copies of a block falls below r_L, it eagerly creates new copies to bring the reachable number up to r_L. Durability is provided by selecting a suitable value of r_L. This parameter must be configured by the user of the system and applied uniformly across all nodes.

In order to maintain at least r_L replicas, Passing Tone allows the number of available replicas to exceed r_L. Because repairs are only executed when fewer than r_L replicas exist, extra replicas serve to mask transient failure and reduce the amount of bandwidth used for transient repairs [13]. To see why, consider the common case where a node is unreachable because of a temporary failure: the node will eventually rejoins the system with its data intact, perhaps after a new replica has been created. These replicas, tracked by Passing Tone, can cause the number of reachable copies of some objects to exceed r_L. The re-use of these replicas, called *replica re-integration*, means that Passing Tone efficiently handles a few hosts that are very unreliable. In such cases, there will be extra redundancy to counter the impact of those hosts: if $r_L + 1$ copies are available, one very unreliable node can join and leave without causing any data to be copied. The replicas beyond r_L act as a buffer that protects against the need to do work in the future. At the same time, the extra replicas provide availability and durability. Nodes only reclaim redundancy beyond the first r_L copies when low on disk space.

The main contribution of this thesis is the design of the algorithms that take this idea and allow it to be implemented in a distributed, efficient manner. To achieve this, Passing Tone removes the need to track object locations explicitly. Instead each node in Passing Tone:

- only keeps a synchronization data structure for objects stored locally;
- shares the responsibility of ensuring adequate replication with the other nodes in the replica set; and
- makes decisions based only on differences detected between itself and an immediate neighbor.

A synchronization protocol is used to determine differences between the set of objects that a node stores locally and the set of objects stored by its neighbors: this represents a set reconciliation problem. As a consequence of keeping only a single synchronization data structure per node, reflecting that node's actual objects, a Passing Tone node must create a replica of objects that it is missing in order to avoid repeatedly learning about that object. The count of available replicas is maintained implicitly as nodes synchronize with their neighbors: when the first r_L nodes in an object's successor list have a replica, there are at least r_L replicas. Passing Tone uses *Merkle trees* for synchronization [12].

Each node in Passing Tone has two maintenance responsibilities. First, it must ensure that it has replicas of objects for which it is responsible. This is its *local maintenance* responsibility. Second, it must ensure that objects it is no longer responsible for but has stored locally are offered to the new responsible node. This represents a node's *global maintenance* responsibility.

```
n.local_maintenance():
  a, b = n.pred_r, n # rL-th predecessor to n
  for partner in n.succ, n.pred:
    diffs = partner.synchronize(a, b)
    for o in diffs:
      data = partner.fetch(o)
      n.db.insert(o, data)
```

Figure 3-1: The local maintenance procedure ensures that each node n has replicas of objects for which it is responsible. The `synchronize(a,b)` method compares the local database with the partner's database to identify objects with keys in the range (a,b) that are not stored locally. Local maintenance does not delete objects locally or remotely.

3.1.1 Local maintenance

Passing Tone's local maintenance ensures that all members of an object's replica set have a replica of the object, despite the fact that this set changes over time. In Passing Tone, the replica set consists of the first r_L successors of the object's key; the underlying Chord protocol ensures that each node has a *successor list* that reflects the current $O(\log N)$ successors of a given node so a given node is always aware of the replica set of objects for which it is directly responsible. This means that Passing Tone need not explicitly track replicas across a replica set because nodes tend to re-enter the replica set after subsequent failures. A node recovering from a transient failure is able to function immediately in the replica set without any repair actions.

Instead of requiring that the immediate successor monitor replicas on other nodes in the replica set, the Passing Tone local maintenance algorithm distributes the maintenance responsibility for a given object over each of the nodes in the current replica set. To distribute responsibility, Passing Tone's local maintenance relies on an extension to Chord that allows each node to know precisely which replica sets it belongs to: our implementation of Chord has each node maintain a *predecessor list* in addition to the successor list. The details of this are discussed in Section 4.4. The predecessor list tells each node the replica sets to which it belongs and hence the range of keys it is responsible for replicating: this allows a node to determine if it needs to replicate an object simply by considering the object's key. By knowing when it is not responsible for an object, the predecessor list also allows a node to discard a replica when it is low on disk space.

The algorithm that implement local maintenance is shown in Figure 3-1. Each node synchronizes with only its direct predecessor and successor, over the range of keys for which it should be holding replicas, as determined by its Chord identifier and predecessors. Any objects that a node is missing are then replicated locally. By making each node responsible for creating replicas that it should have itself, Passing Tone shares responsibility for maintenance of each object over all nodes in the object's replica set: no single node need count the number of replicas of an object. Rather, the nodes in a successor list operate independently but

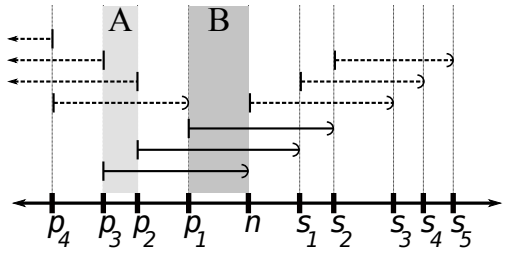


Figure 3-2: Node n is responsible for objects whose keys fall between its predecessor p_1 and itself. In the figure, objects are replicated with an initial replication level $r_L = 3$. Thus, objects that n is responsible for are replicated on s_1 and s_2. The ranges of the objects held by each node is shown with horizontal intervals; the intersection with vertical regions such as A and B show which nodes hold particular objects.

cooperatively to ensure the right number of replicas exist. Despite using only local knowledge, local maintenance does not result in excessive creation of replicas. Because Passing Tone allows objects to remain on nodes even if there are more than r_L total replicas, temporary failures do not cause replicas to be created and then immediately deleted.

By only synchronizing with individual neighbors, Passing Tone ensures that the implementation will never need to maintain or consult information from multiple nodes simultaneously. As a result, memory and disk pressure during maintenance is minimized. This also minimizes the problem that can lead to spurious repairs: instead of accumulating information and referring to it after it has become stale, nodes in Passing Tone make decisions immediately upon synchronizing with the live database of its neighbor.

Synchronizing with the successor and predecessor is sufficient to eventually ensure r_L replicas of any failed objects. This follows directly from how objects are arranged in a Chord consistent hashing scheme. Replicating objects from the successor will allow nodes to recover from missed insertions or disk failures. Similarly, replicating objects from the predecessor will help nodes cover for other nodes that may have failed transiently.

This can be seen more clearly with reference to Figure 3-2. There are two cases to consider that cause object movement: the failure of a node, or the addition of a node. When node n fails, node s_1 becomes responsible for holding replicas of objects in the region labeled A. It can retrieve these replicas from its new predecessor p_1; it is unlikely to retrieve such objects from its successor s_2, though this is possible.

When n is joining the system, it divides the range of keys that its successor is responsible for: n is now responsible for keys in $[p_1, n)$ whereas s_1 is now responsible for $[n, s_1)$. In this case, s_1 is a natural source of objects in the region labeled B. If n was returning from a temporary failure, s_1 will have objects that were inserted when n was absent.

Over time, even if an object is not initially present on the successor or predecessor, it will migrate to those nodes because they themselves are executing the same maintenance protocol. Thus, as long as one node in a replica set has a replica of an object, it will eventually be

```
n.global_maintenance():
  a, b = n.pred_r, n # rL-th predecessor to n
  key = n.db.first_succ (b) # first key after b
  while not between (a, b, key):
    partner = n.find_successor (key)
    diffs = partner.reverse_sync (key, partner)
    for o in diffs:
      data = n.db.read (o)
      partner.store (o, data)
    key = n.db.first_succ (partner)
```

Figure 3-3: Global maintenance ensures objects are placed in the correct replica sets. *n* periodically iterates over its database, finds the appropriate successor for objects it is not responsible for and offers them to that successor. The `reverse_sync` call identifies objects present on *n* but missing on *s* over the specified range.

propagated to all such nodes.

3.1.2 Global maintenance

While local maintenance focuses on objects that a node is responsible for but does not have, global maintenance focuses on objects that a node has but for which it is not responsible. Global maintenance ensures that, even after network partitions or other rapid changes in (perceived) system size, objects are located in a way that DHash's read algorithm and Passing Tone's local maintenance algorithm will be able to access them.

Global and local maintenance in Passing Tone are cooperative and complementary: global maintenance explicitly excludes those objects that are handled by local maintenance. At the same time, global maintenance in Passing Tone relies on local maintenance to correctly propagate replicas to other nodes in the official replica set. That is, once an object has been moved to the successor, the neighbors of the successor will take responsibility for creating replicas of that object across the replica set.

Figure 3-3 shows the pseudo-code for global maintenance. Like local maintenance, global maintenance requires the predecessor list to determine which objects each node should maintain. Node *n* periodically iterates over its database, identifying objects for which it is not in the replica set. For each such object, it looks up the current successor *s* and synchronizes with *s* over the range of objects whose keys fall in *s*'s range of responsibility. *s* then makes a copy of any objects that it is responsible for which it does not have a replica. Once these transfers are complete, global maintenance moves on to consider the next set of objects.

Like in local maintenance, the global maintenance algorithm does not delete replicas from the local disk, even if they are misplaced. These replicas serve as extra insurance against failures. Because the synchronization algorithm efficiently identifies only differences, once the objects have been transferred to the actual successor, future checks will be cheap.

3.1.3 Synchronization

Passing Tone makes use of synchronization to compare the list of objects stored on one node with those stored on another—this determines if there are objects that consistently hash to a given node that the node does not have locally which is important for both global and local maintenance.

Since synchronization occurs fairly often, the efficiency of this comparison is important. In particular, synchronization should be bandwidth efficient and be careful about the amount of local resources (CPU or I/O bandwidth) used on the synchronizing nodes. An inefficient exchange of keys can result in very high bandwidth utilization as the system scales; such an effect was observed in ePost [52] and affected the design of the synchronization mechanisms used in Glacier, ePost's archival storage layer [33]. In the ideal case, the number of keys exchanged during each interval should be no more than the number of differences in the two sets [51].

Similarly, since synchronization is a supporting operation, it should not be so CPU or disk intensive that it significantly affects the ability of the system to do work, especially in environments that are already resource constrained [64]. Thus, any supporting data structures for synchronization should be incrementally updatable and efficient to store/update on disk.

Passing Tone makes use of Merkle synchronization trees and the accompanying network protocol (originally developed by Josh Cates [12]) for synchronization. A synchronization tree is a representation of the keys stored on a node but as a 64-way tree: each node has a hash associated with it representing the concatenation of the hashes of its children. When two sub-trees represent exactly the same set of keys, the hashes of the root of the sub-trees will match and thus that sub-tree can be skipped. Passing Tone uses Merkle trees to compare keys between the successor and predecessor of each (virtual) node.

This thesis develops enhancements to the use and implementation of Merkle trees that enable them to be shared between multiple processes and be stored persistently on disk, without sacrificing the performance of keeping the tree in memory. We discuss these details in Chapter 4.

3.1.4 Expiration support

The expiration time of an object merely acts as a guideline for deletion: a node can hold on to objects past their expiration if there is space to do so. As described above however the local and global maintenance algorithms do not take expiration into account.

Failure to account for expiration in maintenance can lead to spurious repairs. When objects are not expired simultaneously across multiple nodes, which can happen if one node chooses to delete an expired object before its neighbor, the next local maintenance round could identify and re-replicate that object simply because synchronization protocols ignore expiration times. This is undesirable because repairing these objects are a waste of system resources: expired objects are ones that the application has specified as no longer requiring durability.

One solution to this problem would be to include extra metadata during synchronization

so that repairs can be prioritized based on expiration time. Including the metadata initially seems attractive: especially when interacting with mutable objects, metadata may be useful for propagating updates in addition to allowing expiration-prioritized updates. OpenDHT implements a simple variant of this by directly encoding the insertion time into the Merkle tree [63]; this complicates the construction of Merkle trees however and requires a custom Merkle tree be constructed per neighbor.

To address these issues, Passing Tone separates object storage from the synchronization trees. The Merkle synchronization tree contains only those keys stored locally that are not expired. Keys are inserted into the tree during inserts and removed when they expire. Expired objects are removed by DHash when the disk approaches capacity.

By using this separate synchronization data structure for both local and global maintenance, both problems of spurious repairs are avoided, without any complexity related to embedding metadata in the tree itself. The tree remains persistent and can be used to synchronize against any other node.

This expiration scheme assumes that servers have synchronized clocks—without synchronized clocks, servers with slow clocks will repair objects that other servers have already expired from their Merkle trees.

3.2 Design discussion

Passing Tone relies on a number of key parameters and ideas to achieve efficient durability. This section explores the importance of re-integration, considerations for setting the base replication level r_L, and the impact of placement policy.

3.2.1 Context

We consider Passing Tone in three different deployment scenarios, summarized in Table 3-2. Scenario A considers a deployment on the PlanetLab distributed test-bed [59]. This is a large test-bed with limited per-node capacity but whose transient and permanent failure patterns, as a platform aimed at hosting distributed services, are demonstrative of typical DHT deployments. Scenario B considers the RON test-bed [2], which consists of fewer nodes but where each one has better transfer and storage capacity. Finally, as Scenario C, we consider a hypothetical deployment with high transfer and storage capacity.

Evaluation of Passing Tone in PlanetLab (Scenario A) considers a trace of transient and permanent failures over a one year period. This trace was originally published as part of the evaluation of Carbonite [13]. The trace consists of a total of 632 unique hosts experiencing 21,255 transient failures and 219 disk failures. Failures are not evenly distributed, with 466 hosts experiencing no disk failures, and 56 hosts experiencing no disk or transient failures. We reproduce a summary of its salient characteristics in Table 3-1.

The Carbonite study used historical data, collected by the CoMon project [56], to identify transient failures. CoMon has records collected on average every 5 minutes that include the uptime as reported by the system uptime counter on each node. The Carbonite study

Dates	1 Mar 2005 – 28 Feb 2006
Number of hosts	632
Number of transient failures	21255
Number of disk failures	219
Transient host downtime (s)	1208, 104647, 14242
Any failure interarrival (s)	305, 1467, 3306
Disk failures interarrival (s) (Median/Mean/90th percentile)	54411, 143476, 490047

Table 3-1: CoMon+PLC trace characteristics.

used resets of this counter to detect reboots, and estimated the time when the node became unreachable based on the last time CoMon was able to successfully contact the node. This allowed transient failures to be pinpointed without depending on the reachability of the node from the CoMon monitoring site.

A permanent failure is defined to be any permanent loss of disk contents, due to disk hardware failure or because its contents are erased accidentally or intentionally. In order to identify disk failures, the Carbonite study supplemented the CoMon measurements with event logs from PlanetLab Central [58]. PlanetLab automatically records each time a PlanetLab node is reinstalled (e.g., for an upgrade, or after a disk is replaced following a failure). The machine is then considered offline until the machine is assigned a regular boot state.

Our actual deployment makes use (in part) of the Emulab wide-area test-bed, originally developed for the RON project [2] and also nodes provided by colleagues at research institutions. Unlike PlanetLab, which limits the per-node bandwidth utilization to an average of 150 Kbyte/s and disk usage to 5 Gbyte, the wide-area nodes in RON have much higher bandwidth and much more disk capacity: one node has a connection that supports transfers at nearly 10 Mbyte/s, and each node used in RON has on average 120 Gbyte of storage. These nodes are also more reliable than PlanetLab nodes: in 2006, PlanetLab event data indicates 219 instances where data was lost due to disk failure or upgrade whereas no such failures were recorded for the wide-area nodes used. The primary limitation of this test-bed is that there are only seven sites with similarly equipped nodes available. This class of deployment is Scenario B.

We believe that wide-area nodes that will ultimately run such large infrastructure-type applications will be more numerous and better provisioned in disk than our current nodes. Thus Scenario C imagines parameter choices for a system with one hundred nodes, T3 connections and 1 Tbyte of storage.

3.2.2 Importance of re-integration

Like Carbonite, Passing Tone re-integrates object replicas stored on nodes after transient failures. This reduces the number of transient failures to which Passing Tone responds, relative to a system that does not re-integrate. For example, if the system has created two replicas beyond r_L and both fail, no work needs to be done unless a third replica fails before one of

Scenario	Avg. Population	Per-node Storage	Per-node Bandwidth
A (PlanetLab)	490	5 Gbyte	150 Kbytes/s (T1)
B (RON)	12	120 Gbyte	1 Mbyte/s
C (Hypothetical)	100	1 Tbyte	5 Mbyte/s (T3)

Table 3-2: Sample networks for understanding the impact of different parameter choices.

the two currently unavailable replicas returns. Once enough extra replicas have been created, it is unlikely that fewer than r_L of them will be available at any given time. Over time, it is increasingly unlikely that the system will need to make any more replicas.

The number of replicas that the system creates depends on a, the average fraction of time that a node is available [14]. It is easy to conclude that this means the system must be able to track more than r_L replicas of each object. This is not correct: Passing Tone does not explicitly track the locations of these replicas. Additionally, the system still creates the correct number of extra replicas without estimating a (as is done in Total Recall [6]). The bandwidth impact of re-integration can be seen below in Section 3.3: Passing Tone uses 77% of the bandwidth required by the naïve maintenance scheme without re-integration used in CFS.

3.2.3 Setting replication level

The critical parameter for durability in Passing Tone is r_L. Unfortunately, it is impossible to know how much replication will be absolutely needed to provide durability for a given length of time. There is a non-zero (though small) probability for every burst size up to the total number of nodes in the system. Moreover, a burst may arrive while there are fewer than r_L replicas. Thus, no value of r_L, no matter how high, will guarantee durability.

A system designer must make an educated guess so that, for a given deployment environment, r_L is high enough that the probability of a burst of r_L failures is smaller than the desired probability of data loss. We find that for most server applications $r_L = 3$ is more than adequate for durability and even $r_L = 2$ works well for some scenarios.

One approach might be to set r_L to one more than the maximum burst of simultaneous failures in a trace of a real system. To determine what is consider simultaneous, we use the time it might take for a single node to make a copy of its disk—the disk capacity divided by the available bandwidth. For example, Figure 3-4 shows the burstiness of permanent failures in the PlanetLab trace by counting the number of additional permanent failures that occur in the 12, 30 and 60 hours following a given permanent failure; the total number of failures in that period are counted. If the size of a failure burst exceeds the number of replicas, some objects may be lost. From this, one might conclude that as many as 14 replicas are needed to maintain the desired durability. This value would likely provide durability but at a high cost. However, while Figure 3-4 shows that bursts of fourteen failures can happen within a three day window, there is no reason to suspect that all of those failures affect disks in the same replica set. In fact, the choice of random identifiers in Chord helps ensure that nodes that might fail concurrently (e.g., located in the same machine room), are not part of the same

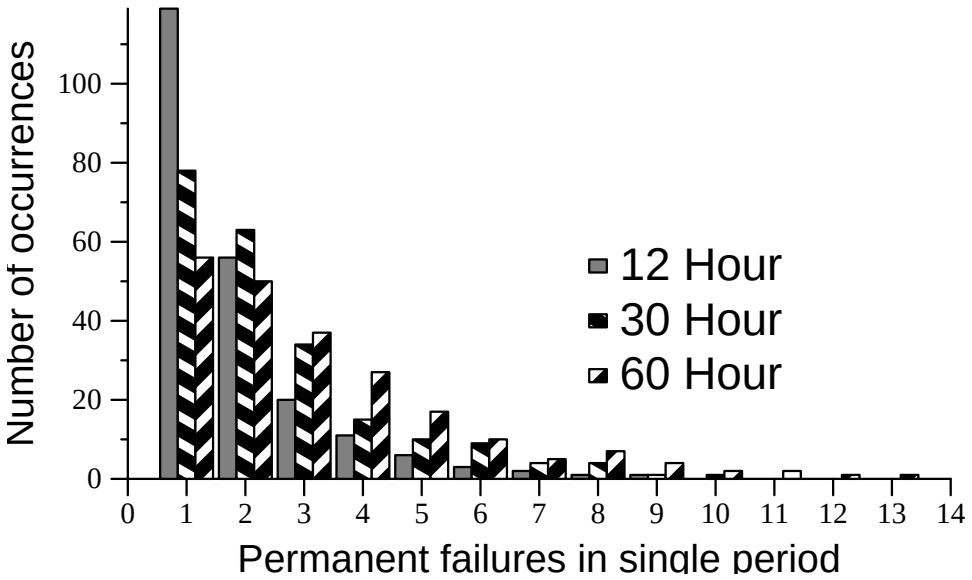


Figure 3-4: Frequency of simultaneous permanent failures in the PlanetLab trace. These counts are derived by considering a period of 12, 30 and 60 hours following each individual crash and noting the number of permanent failures that occur in that period. These choices correspond roughly to the time for a single node to duplicate its data on another node in scenarios A, B, and C. If there are x replicas of an object, there were y chances in the trace for the object to be lost; this would happen if the nodes with remaining replicas were not able to respond quickly enough to create new replicas of the object.

replica set (i.e., are not neighbors in the identifier space). Since a lower value of r_L would suffice, the bandwidth spent maintaining the extra replicas would be wasted.

Another approach for setting r_L comes from the Glacier system. The Glacier storage layer for ePost defines durability as "the probability that a specific data object will survive an assumed worst-case system failure" [33]. Glacier calculates the amount of redundancy needed to reach a target durability level after catastrophic, Byzantine failures. We are not considering catastrophic failures but simply coincident random failures; instead of surviving a failure of $f \approx 60\%$ of all nodes, we might consider a value of $f = 5\%$. Under the Glacier model and using only replication, a target of survival five nines after the failure of $f = 5\%$ of the nodes, would require $r_L = 4$.

We found that, based on simulations run against the trace for scenario A, three replicas are needed for durability. This correlates reasonably well with a failure of 14 nodes in a system that averages 490 nodes ($f \approx 3\%$). On the other hand, our expected deployment scenarios will be able to make replicas faster than the PlanetLab nodes can; in such cases, it may well be that a replication level of two is sufficient to provide 4 or more nines of durability.

3.2.4 Scope and parallelism

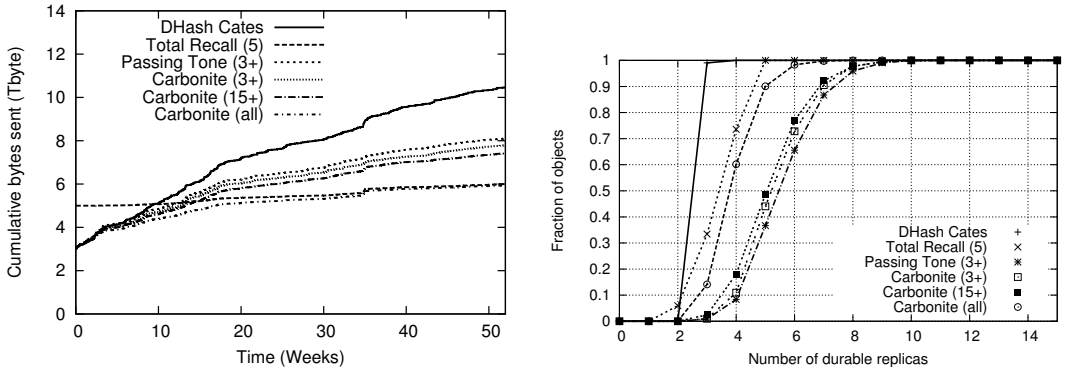
The value of r_L needed to provide a certain level of durability depends not only the rate at which permanent failures arrive, but how replicas are stored on nodes in the system. DHT placement algorithms specify, for each node n, a set of other nodes that can potentially hold copies of the objects that n is responsible for. We will call the size of that set the node's *scope*, and consider only system designs in which every node has the same scope. Scope can range from a minimum of r_L to a maximum of the number of nodes in the system. The choice of placement across the nodes in scope affect the efficiency of read and write operations, and thus also the speed with which the system can make repairs. Placement also affects the likelihood that a given series of permanent failures will cause the loss of an object.

A small scope means that all the objects stored on node n have copies on nodes chosen from the same restricted set of other nodes. The disadvantage of a small scope is that the effort of creating new copies of objects stored on a failed disk falls on the small set of nodes in that disk's scope. The time required to create the new copies is proportional to the amount of data on one disk divided by the scope. Thus a smaller scope results in a longer recovery time.

Larger scopes spread the work of making new copies of objects on a failed disk over more access links, so that the copying can be completed faster. In the extreme of a scope of N (the number of nodes in the system), the remaining copies of the objects on a failed disk would be spread over all nodes, assuming that there are many more objects than nodes. Furthermore, the new object copies created after the failure would also be spread over all the nodes. Thus the network traffic sources and destinations are spread over all the access links, and the time to recover from the failure is short (proportional to the amount of data per disk divided by N).

The impact of a particular set of failures depends also on the size of the scope and placement. For a system with N nodes storing r_L replicas, there are $\binom{N}{r_L}$ potential replica sets. If objects are placed randomly with scope N and there are many objects, then it is likely that all $\binom{N}{r_L}$ potential replica sets are used. In this scenario, the simultaneous failure of any r_L disks is likely to cause data loss: there is likely to be at least one object replicated on exactly those disks. A small scope limits placement possibilities that are used, concentrating objects into common replica sets. As a result, it is less likely that a given set of r_L failures will affect a replica set, but when data loss does occur, many more objects will be lost. These effects exactly balance: the expected number of objects lost during a large failure event is identical for both strategies. It is the variance that differs between the two strategies.

It is clear that larger scopes result in more parallelism. However, as the placement scheme considers more and more nodes, implementation becomes challenging, as noted in Section 2.2.3. Passing Tone limits the scope but takes advantage of a node's virtual nodes—each virtual node communicate only with its immediate neighbors but because each virtual node manages only a fraction of the node's disk, there is additional parallelism gained. This is not as fast as a scope of N would allow; however, this choice simplifies the implementation significantly and still allows the system a measure of higher repair parallelism.



(a) (b)

Figure 3-5: A comparison of several maintenance algorithms with $r_L = 3$ over the PlanetLab trace. The total amount of bytes sent over time (a) and the cumulative distribution of the resulting number of replicas at the end (b) is shown. In all cases, no objects are lost. The legends indicate the algorithm used and the number of replica holders that the algorithm is capable of tracking. For example, "3+" indicates that the algorithm explicitly tracks replicas on the first three successors but will re-use replicas on subsequent successors if one of the first three fail. The graphs highlight the effectiveness of replica re-integration; even without the machinery of Carbonite, Passing Tone sends only 3.6% more bytes than Carbonite limited to three nodes.

3.3 Evaluation

We evaluate Passing Tone relative to other algorithms—in particular, its ability to provide durability—by considering its behavior using a trace-driven simulator over the PlanetLab trace described above and summarized in Table 3-1. At the start of the trace, 50,000 20 Mbyte objects are inserted and replicated according to standard DHash placement. With an average of 490 online servers and $r_L = 3$ replicas, this corresponds to just over 5 Gbyte of data per server. To approximate PlanetLab bandwidth limits, each server has 150 Kbyte/s of bandwidth for object repairs. At this rate, re-creating the contents of a single server takes approximately 12 hours.

Figure 3-5 shows the behavior of Passing Tone on this trace, and compares it to several prior DHT systems. In the simulation, each system operates with $r_L = 3$; that is, no repairs are initiated until fewer than three live replicas remain for a given object. The systems are simulated against the PlanetLab trace. Figure 3-5(a) plots the cumulative bytes of network traffic used to create replicas against time. Figure 3-5(b) plots the cumulative distribution of the number of replicas in existence at the end of the trace. The algorithms compared largely differ in the number of nodes actively monitored for a given object, and how the algorithms track of replicas that may be offline.

For Passing Tone to be viable, it must not lose any objects. Figure 3-5(b) shows a CDF of the number of replicas for each object at the end of the trace: the most important feature is that all objects have at least three replicas. No objects are lost, showing that Passing Tone can

provide durability in a PlanetLab environment.

In comparison to Passing Tone, the original DHash Cates algorithm actively tracked only replicas on the first three successors and discarded the remainder: as a result, there are precisely three replicas in existence for every object at the end of the trace. On the other hand, this algorithm must constantly create and discard replicas as nodes transiently fail and return, resulting in the highest bandwidth usage of any algorithm.

Total Recall tracks more than only the first three replicas; however, like DHash Cates, it restricts itself to tracking a fixed number of replicas r_H by explicitly remembering the locations of replicas for each object. It initially inserts a larger number of replicas in a batch to attempt to defer repair action. When repair does occur, it forgets the existence of any offline replicas since only r_H replicas can be tracked. This makes Total Recall very sensitive to settings of r_H. The graph shows a value of $r_H = 5$ which, for this particular trace, does not lose any objects. However, the CDF demonstrates that some objects have only one replica remaining.

If r_H is set too low, a series of transient failures will cause the replication level to drop below r_L and force it to create an unnecessary copy. This will cause Total Recall to approach Cates (when $r_H = r_L$). Worse, when Total Recall creates new copies it forgets about any copies that are currently on failed nodes and cannot benefit from the return of those copies. Without a sufficiently large memory, Total Recall must make additional replicas. On the other hand, setting r_H too high imposes a very high insertion cost and results in work that may not be needed for a long time.

Carbonite and Passing Tone dynamically track replicas across many nodes. Passing Tone synchronizes with either its predecessor or successor to generate repairs servers change; each Carbonite node periodically synchronizes with the other nodes it monitors. Both explicitly tracks any replicas across a fixed number of successor nodes (denoted in parentheses in the graph legends) but does not forget or delete extra replicas by default so that extra replicas will be re-incorporated or re-used after transient failures. The number of replicas for each object varies, depending on how stable the hosts of that particular object are. Dynamic tracking and flexible replication levels allow Carbonite and Passing Tone to use less bandwidth than prior systems: Passing Tone uses 77% of the bandwidth used by DHash Cates.

In the ideal (though essentially impossible to implement) case, Carbonite tracks replicas across *all* hosts: very few additional replicas are needed in this case as joins and most failures can be ignored. Passing Tone responds initially to all transient failures, visible in Figure 3-5(a) as Passing Tone builds up a buffer of extra copies. More feasible (though still difficult) cases of Carbonite involve tracking fewer nodes (e.g., 15 or even 3): this means that some copies are not accessible and nodes make more copies.

Passing Tone is most similar to a Carbonite configuration explicitly tracking only three nodes. The bytes sent differ only by 3.6%; this small difference is due to the fact that Passing Tone distributes responsibility for a given object across its replica set. Unlike Carbonite, there is no central node that can prioritize replication of objects based on the number of available replicas.

Passing Tone and Carbonite (when restricted away from total scope), both create more than three replicas. The CDF demonstrates that ten percent have over seven replicas. To

understand why, consider Figure 3-4, which shows the number of times a given number of servers crashed (i.e., lost disks) within an 12 hour period over the entire trace; the worst case data loss failure in the trace could only be protected if at least nine replicas existed for objects on the nine simultaneously failing servers. It is unlikely that the actual failure in the trace would have mapped to a contiguous set of servers on a Chord ring; however, the simulation shows that Passing Tone saw sufficient transient failures to create nine replicas in at least some cases.

In summary, Passing Tone offers a usable option between the multi-node coordination and synchronization required by Carbonite and the limited memory and poor performance of other earlier schemes. Automatic re-use of existing replicas is an important part of Passing Tone's design. The resulting algorithms are simple and easy to implement in distributed systems.

Chapter 4

Implementation

The implementation of Chord/DHash and Passing Tone has been driven by a number of goals. First, as motivated throughout this thesis, we desire the implementation to demonstrate the ability of the algorithms to support high-performance read/write/expiration workloads. Second, refinement of these algorithms proceeded in parallel with their implementation; thus the implementation has aimed to provide a degree of flexibility in supporting different alternatives for almost all aspects of the system from on-disk storage to routing to maintenance.

The implementation, including all supporting libraries and applications, comprises some 41,000 lines of C++ code written using the `libasync` asynchronous programming libraries [50]. A breakdown of this code by module is given in Table 4-1. The bulk of the code is in supporting infrastructure—the local and global maintenance code for Passing Tone itself is under 1,000 lines of code.

Much of the complexity in the routing and other code comes from the goal of supporting multiple alternatives for common functions including RPC transports, DHT routing, DHash object types and maintenance algorithms. Other complexity is due to the need to support I/O parallelism: because of the need to access local disk efficiently, DHash is structured into multiple processes in order avoid having blocking disk I/O calls prevent DHash from responding to unrelated network requests. The main location service daemon `lsd` process receives network RPCs and translates disk I/O requests to calls to `adbd`, a separate database process that operates asynchronously. Maintenance is handled by `maintd`, a separate process that shares access to the on-disk storage and handles network and disk I/O related to maintenance.

The remainder of this chapter discusses notable aspects of the implementation relating to performance and also describes particular problems that were encountered and solutions that were developed in order to solve those problems.

4.1 DHash

DHash performs the core storage function of a distributed hash table. To do this effectively, DHash must both offer high-performance disk storage but also efficient network transfers.

Module	LoC	Description
`utils`	12500	Utility functions and data structures, including BerkeleyDB common code and IDA [61] (with a 7000 line generated file for $\mathbb{Z}_{65537}$ math).
`chord`	7967	Chord lookup and routing client and server implementation [78], including the RPC and transport functions with performance enhancements [17] and support for Accordion [46].
`dhash`	5166	Storage client and server implementation, including support for different object types.
`maint`	1759	Implementations of Passing Tone and Carbonite [13].
`merkle`	5037	Merkle synchronization trees and network protocol implementation [12].
`lsd`	2820	Main server front-end and on-disk database management back-end.
`tools`	6239	Command-line tools for testing, querying, and monitoring routing and storage behavior.

Table 4-1: Source code overview by module, including lines of source code (using `wc`) and brief descriptions. Line counts include test code.

The current deployment of DHash relies on both BerkeleyDB and regular files to manage data and metadata; TCP is used for data transfer.

4.1.1 Process structure

Standard Unix system calls for disk I/O are blocking. Because `libasync` is single-threaded blocking system calls caused a significant performance penalty since it prevents a DHash node from serving DHash requests and routing independent Chord requests simultaneously,

In order to allow the operating system to overlap I/O scheduling with CPU activity, storage (and maintenance) related disk I/O is separated from the main daemon process, `lsd`, into a database process, `adbd`, and a maintenance process, `maintd`. The DHash objects in `lsd` initialize corresponding storage and maintenance objects in `adbd` and `maintd` and proxy all *put*/*get* requests into RPCs that asynchronously execute in the I/O helper daemons.

One area however where we do want blocking behavior is during node startup. Early versions of DHash would come online without having fully initialized the disk databases—for example, `adbd` may have exited uncleanly and have to perform an expensive database recovery. Until this completes, the DHash side of the implementation would not be ready to service incoming requests. However, if the Chord side of `lsd` was already advertising the node's availability, requests could come in that would either be rejected or delayed resulting in errors. To prevent this, it is important to ensure that *no* network RPCs are initiated for any virtual node until all of them have completed the necessary initialization.

4.1.2 Storage

`adbd` stores objects and metadata separately. Flat files allow `adbd` to add new objects efficiently (via append) and read existing objects without extra seeks. Grouping multiple objects into a single file allows efficient reclamation of expired data when the disk approaches capacity. BerkeleyDB databases provide a simple key-value store, that `adbd` uses to map object keys to metadata, such as expiration time, object size and synchronization trees.

Storing objects outside of BerkeleyDB is important for performance. Since BerkeleyDB stores data and key within the same BTree page and since pages are fixed size, objects larger than a single page are stored on separate overflow pages. This results in particularly poor disk performance as overflow pages cause writes to be chunked into page-sized fragments that are not necessarily located contiguously, leading to additional disk seeks.

For metadata, BerkeleyDB's transaction system is used to provide atomicity, consistency and isolation between processes. BerkeleyDB allows the log to be flushed asynchronously to disk for improved performance, without sacrificing on-disk consistency. This results in the potential for data loss during crashes; Passing Tone maintenance will detect the loss of these replicas and repair them.

`adbd` names object storage files by approximate expiration time. This idea is similar to the timehash method used in the INN news server [71]. The upper 16 bits of a 32-bit Unix time stamp (seconds since epoch) are masked and formatted in hexadecimal to create a directory; the next lower 8 bits are similarly used to name a file. Each file holds 256 seconds worth of objects, which works well for the case of continuous writes where expiration times are usually at a fixed offset from the insertion time. In Usenet, for example, the worst case might be a single host taking a full feed which would result in 50 173 Kbyte writes per second or an approximately 2 Gbyte file. Of course, in UsenetDHT, this load would be spread out over significantly more nodes.

DHash does not support the idea of explicit deletes. Instead, motivated in part by UsenetDHT, DHash provides expiration. Our early implementation of expiration enumerated objects in order of their expiration time and then removed them individually. This led to significant delays as each removal required numerous disk seeks, for updating the overflow pages, their associated metadata and then rebalancing the trees.

4.1.3 Efficient data transfer

Transport. The original design of DHash and its data transfer mechanism relied on TCP. DHash was originally built for file-system applications such as CFS [16] so all the objects transferred were small (e.g., 8 Kbyte blocks). Combined with a vision to scale to millions of nodes, this meant that TCP would not be a suitable protocol for transport. First, these usage patterns meant that a given node would typically be talking to another node that it had either never interacted with previously, or that it had not interacted with in a long time. Thus, to establish a TCP connection would require the delay of a 3-way handshake. Second, because interactions with particular remote nodes are infrequent (especially compared to, say, the immediate successor) and objects transferred are small, each TCP connection would see

poor throughput as it would typically remain in slow-start. Additionally, early experience with large number of nodes led to implementation issues such as running out of active file descriptors as TCP connections were held open indefinitely.

For these reasons, DHash moved to using a custom transfer protocol called STP [17]. STP offered several advantages over individual TCP connections: packets were sent over UDP and congestion-controlled fairly across multiple hosts. STP worked well with the structure of DHash objects which were then erasure-coded into fragments that could be sent within a single Ethernet MTU-sized packet. STP used Vivaldi synthetic coordinates [15] to allow DHash nodes to estimate the round-trip time to new nodes and select those that had the lowest round-trip times from which to read. Dabek *et al* demonstrated that this protocol achieved high throughput relative to TCP, while remaining TCP-friendly [14].

Large objects did not work well with STP: larger object sizes meant that DHash had to chunk objects to avoid fragmentation (since fragmented packets are often dropped by over-cautious firewalls). However, applications like UsenetDHT, OverCite and CFS that motivate DHTs are more likely to be server-driven and thus involve fewer hosts. In these cases, it would be faster to directly transfer whole articles, as opposed to retrieving numerous sub-objects from many nodes.

Thus, UsenetDHT motivated a move away from using STP and storing single 8 Kbyte objects, towards larger objects and TCP. In addition to higher transfer throughput performance, storing large objects removes the need to have book-keeping objects that track the many DHT objects comprising each Usenet article. DHash now uses TCP connections for data transfer: individual transfers are larger, avoiding the penalty of short slow-start constrained connections, and there are few enough hosts that it is unlikely that we will have implementation issues related to file descriptors. Server selection via synthetic coordinates also becomes less important; in this situation, BitTorrent-style connection ranking would be a natural option for improving overall throughput but this option has not yet been explored.

Caveats and details. To avoid liveness detection problems (see Section 4.4 below), it is important to use TCP for *all* data transfers. This prevents confusion about node state when STP/UDP RPCs are dropped while TCP RPCs continue to flow. For example, a long-latency RPC executed remotely may cause the STP retransmit timer to expire (possibly multiple times, since some RPCs can take seconds to execute due to disk scheduling [64]) and ultimately be treated as a node failure. However, the node would still be alive and there may still be an active TCP connection to that node. This problem could be fixed by adding an explicit RPC acknowledgment. However, we have not taken this step. By using solely TCP this problem is avoided. While it is possible to use STP for regular routing table maintenance though there is no particular reason to.

The current implementation maintains a cache of TCP connections and closes them when a given connection has been idle for several minutes. In practice in our test-beds, this leaves most connections long-running and able to instantly support traffic.

For historical reasons related to disk I/O, the current DHash fetch protocol has a client make a FETCHITER RPC to a remote peer, receive an "in-progress" response and then at

some point later, receive the actual object in a series of FETCHCOMPLETE RPCs initiated by the remote peer. This originated as a solution to problems caused by the lack of an explicit acknowledgment of RPC receipt, which can cause RPCs to appear lost (and thus cause retransmissions or failures) when disk I/O delays affect RPC delays and re-transmission timeouts. With strict TCP, this protocol could be streamlined.

Alternatives. While TCP does offer good performance and generally useful semantics, the DHT application does not compete fairly with other single-TCP stream applications. TCP Nice could be a valid way to deal with the fact that multiple parallel TCP connections compete differently [85]. Of course, many other popular Internet applications, such as web browsers and file-sharing tools, also make use of multiple parallel TCP connections.

Another problem is that regular routing table maintenance may be blocked by large object transfers, since TCP offers only a single buffer between hosts. A recent alternative that addresses this problem is the Structured Stream Transport, which removes head-of-line waiting and other problems associated with straight TCP for logically separate RPC streams [20].

A more extreme approach might be to completely replace the data transport system with the Data-Oriented Transfer architecture [82]; this system can be used to separate data transfer from content location. We have not observed the downsides that might make these alternatives attractive and thus not moved towards implementing them.

4.2 Maintenance

Supporting experimentation with maintenance protocols required separating maintenance and the generation of repairs from DHash itself. Thus the `maintd` process is charge of executing an appropriate maintenance scheme and providing repairs to DHash. A *repair* consists of an object identifier, a destination node, and an optional source node.

There is a tension between flexibility in the ability to handle different object placement strategies and the simplicity of the code. Random placement of objects (and replicas) versus consistent hashing placement may require different maintenance strategies that do not map well to this structure; the current implementation supports only placement of objects on (potentially random) nodes within the successor list.

In order to identify repairs, `maintd` queries the local `lsd` for information about the current successors (in order to identify potential candidates for holding replicas) and predecessors (in order to identify the correct range of keys for which it should hold replicas). `maintd` then examines the local database, synchronizes with neighbors provided by `lsd` to make a determination about what objects need to be repaired. Asynchronously, `lsd` periodically requests repairs from `maintd`, performing a standard read to obtain the object and then storing it on the destination node. For Passing Tone, this means reading the object from a specified neighbor and then storing it locally.

Expiration. Node failures and joins present a risk for spurious repairs. The internal queue that is processed on a per-repair basis by `lsd` can accumulate repairs significantly faster than the repairs themselves can be completed. Two risks result from having many repairs enqueued. First, objects may remain on the queue even after some redundancy returns—a short term failure of two nodes may cause repairs to be queued but the return of either of those nodes could obviate the need for repair. Second, objects may remain on the queue and be processed after their expiration time. Such repairs are unnecessary and would better be avoided.

The first problem is avoided by flushing the queue whenever any change in the neighbor set is detected. This corrects problems due both to range of responsibility differences (changes in predecessor list) and in replication level (successor list changes).

The second problem is more difficult to handle correctly. One work-around would be to avoid queuing objects faster than can be processed (or capping the number of objects that can be queued). However, the implementation of Merkle synchronization does not lend itself to operating in a manner that allows an in-progress synchronization to be canceled or paused. The semantics of synchronization over a changing set (on both sides) is also not well defined.

Another approach would be to special-case repairs of broad key ranges, perhaps in a way similar to how Amazon's Dynamo breaks the whole identifier space into discrete ranges [19]. Node joins and failures would result in a concrete number of ranges that would need to be sent over to new nodes in their entirety: this could greatly reduce the number of disk seeks required to recover from a node failure, improving replication performance, and make the problem much less likely to occur.

The most precise approach would be to include each object's expiration time as part of the enqueued repair. This could be accomplished by including the expiration time as part of the synchronization protocol, which would require re-working tree construction to embed the expiration time as part of the keys inserted into objects. This approach is adopted by OpenDHT [63] to prioritize newer objects. An implementation in DHash would store the expiration time in the low-order bits of the embedded key instead of the higher-order bits; this would avoid OpenDHT's need to construct trees on the fly for each remote node and preserve the ability to limit the synchronization to objects in a particular key range.

Repair priority. If repairs interfere with normal operation (e.g., by competing for the disk arm), it is not clear how the conflict should be resolved. Our current approach is simply to let the repairs proceed, possibly limiting the ability of a node to process normal requests. Amazon's Dynamo takes a more nuanced approach of lowering the priority of rebalancing and repair requests when the load is high, favoring the ability of the system to handle real workload and relying on the relative infrequency of failures to preserve durability.

4.3 Synchronization

Storage lessons. Merkle trees were originally designed so that all internal nodes would be stored in memory ephemerally, and recalculated at startup from the actual objects stored on disk. However, BerkeleyDB can not efficiently enumerate the keys of the database; each page

read returned few keys and much data. Further, BerkeleyDB does not include an API for enumerating the database in on-disk order. A straightforward reconstruction of the Merkle tree requires one seek per key over the entire set of keys held in the tree. With a 120 Gbyte disk (holding 600,000 173 Kbyte articles), at 5ms per seek, enumerating these keys could take over 50 minutes.

When `lsd` was split into multiple processes, the first approach was to construct two in-memory Merkle trees. Since this split was motivated by avoiding disk I/O, the key lists themselves were moved into memory. This used up available physical memory when each node was run with multiple virtual nodes each holding millions of objects.

Persistently storing the Merkle tree enables sharing and provides good start-up performance. `adbd` currently uses a pair of databases to store a Merkle tree. `adbd` handles write requests and updates the Merkle tree as new objects are written and old ones are expired. `maintd` reads from this same tree during synchronization. BerkeleyDB's transaction system is used to provide atomicity, consistency and isolation between databases and between processes. For performance, the implementation disables writing the transaction log synchronously to disk.

The Merkle tree implementation currently stores the pre-computed internal nodes of the Merkle tree in a separate database addressed directly by their prefix. Prefix addressing allows `maintd` to directly retrieve internal nodes during synchronization without traversing down from the root. `maintd` uses a BerkeleyDB internal in-memory cache, sized to keep most of the nodes of the Merkle tree in memory, retaining the performance benefits of an in-memory implementation. The implementation stores object keys in a separate database. Since keys are small, the backing database pages, indexed by expiration time, are able to hold dozens of keys per page, allowing key exchanges in Merkle synchronization to be done with few disk seeks as well. Being indexed by expiration time, it is easy to update the Merkle synchronization tree to remove keys as they expire.

Our initial implementation in BerkeleyDB experienced a significant number of internal deadlocks under high load. Investigation revealed that these were caused by the locks that had to be acquired to re-balance pages during key deletions (driven by constant expirations). Since we are in a high write workload situation, we disabled reverse splits (i.e., page coalescing) as subsequent insertions quickly require these pages to be reallocated.

Correctness under mutation. The synchronization of a node's objects with another's, using the Merkle synchronization protocol, can take seconds or minutes to complete. Because of the time required, the Merkle client or server may by presented with network messages that reflect past state that is inconsistent with the current tree. That is, in an active system, it is unlikely that any two nodes would receive no writes for a period long enough to completely traverse their synchronization trees. As a result, Merkle protocol messages may occasionally refer to Merkle tree nodes have changed relative to the state that was exchanged earlier during the traversal. The Merkle synchronization implementation checks for inconsistencies during execution and ignores portions of the keyspace that are inconsistent with previous messages; these portions can be re-tried later to allow the system to eventually become consistent.

4.4 Routing and node management

Our implementation supports multiple different DHT routing algorithms including Koorde [36], Accordion [46], as well as many variants of Chord [78]. These variations were largely focused on performance and include techniques such as recursive routing, proximity neighbor selection for fingers (via synthetic coordinates maintained using the Vivaldi algorithm [15]), and successor list assembly. We describe the APIs and the evaluation of these variations in our prior work [17, 18] and the details are discussed by Dabek [14].

Lookup. The base Chord lookup primitive returns the immediate successor of a key. However, the needs of DHash are slightly different—for `put`/`get` operations, DHash needs the list of the first r_L active successors: more than the immediate successor and less than $O(\log N)$ nodes in the entire successor list of the actual successor. By communicating this more precise need to the routing layer, DHash provides the routing layer with the option of choosing between several nodes even as the successor gets closer. With options, a low-latency node can be selected, instead of requiring that the predecessor (which has an average latency) be contacted.

DHash filters the resulting successor list to select only the first virtual node for a given physical node. A node can appear multiple times when, due to randomness or a paucity of nodes, its virtual nodes are adjacent or nearly adjacent in the identifier space. By removing such duplicate virtual nodes, DHash ensures that all RPCs that are then issued, whether for read or write, are issued to each physical node only once. Since all virtual nodes share the same logical database, this does not affect correctness. More importantly, it helps ensure even load balance over the disk spindles available to each physical node; a node's spindles are usually under high contention as typically a physical node will have more virtual nodes that physical disk spindles.

Predecessor lists. In performing maintenance, Passing Tone requires information about the objects for which each individual node is responsible. This requires that each node know its first r_L *predecessors*. Predecessor lists are maintained using the inverse of the Chord successor list algorithm. Each node periodically asks its current predecessor for their predecessor list. When providing a predecessor list to a successor, a node takes its own predecessor list, shifts off the furthest entry and pushes on itself.

Predecessor lists are not guaranteed to be correct. Nodes are not considered fully joined into the Chord ring until they have received a notification from their predecessor. The predecessor maintenance algorithm may return a shortened, wrong or empty predecessor list in this situation. For this reason, this algorithm was rejected in the design of Koorde [36], which relies on predecessor lists for correct routing. However, this situation occurs fairly rarely in practice, if at all, and we have never observed any problems resulting from this implementation. Because Passing Tone does not rely on predecessor lists for routing, we accept the occasional possibility of error. Even in the event that an incorrect predecessor list is used for determining maintenance, Chord stabilization will soon cause the predecessor list

to change and any outstanding incorrect repairs will be flushed before significant bandwidth has been used.

Node state. An important aspect of the implementation is maintaining information about the mapping between Chord identifiers and network addresses (IP address and UDP/TCP port numbers) that is necessary for communication. One early goal of the Chord design was to minimize the number of other nodes in the system that were known about. However, because the maintenance of the internal node list (called the `locationtable`) happened asynchronously to on-going actions, this could result in the eviction of information about remote nodes that a local was node was still interacting with. The implementation now explicitly passes the identifier to network address mapping to functions in reference-counted `location` objects that are only garbage-collected when all communication has completed.

Liveness detection. Various deployments of our implementation were plagued by a problem of nodes that were dead but would not fade from the successor lists of various nodes, despite changes to have each node individually verify the reachability of other nodes. Because nodes would gossip information about other nodes, misleading information could propagate indefinitely.

We adopt a solution from Accordion's design: when gossiping information, each node also includes the age of that node information and the last time the node itself successfully communicated with the remote node. The node receiving the information can then filter older information. If the local state indicates that a node is dead, that can override information received remotely that indicates that a node is alive if the local state is newer. Over time, this ensures that stale nodes are excised from the system.

Accordion's solution does not deal with the problem of asymmetric reachability which can occur for policy reasons (e.g., a network with a mixture of Internet2-only nodes and non-Internet2 nodes) or due to routing errors. The implementation currently does not deal well with network routing asymmetry.

Chapter 5

Application: UsenetDHT

UsenetDHT provides a Usenet server that reduces aggregate bandwidth and storage requirements relative to traditional servers. UsenetDHT is designed to be operated by and among mutually trusting organizations that can cooperate to share storage and network load. Prime examples of such organizations are universities, such as those on Internet2, that share high-bandwidth connectivity internally and whose commercial connectivity is more expensive. For such organizations, UsenetDHT aims to:

- reduce bandwidth and storage costs in the common case for all participants;
- minimize disruption to users by preserving an NNTP interface; and
- preserve the economic model of Usenet, where clients pay for access to their local NNTP server and can publish content without the need to provide storage resources or be online for the content to be accessible.

UsenetDHT accomplishes these goals by organizing the storage resources of servers into a shared distributed hash table (DHT) that stores all article data. A front-end that speaks the standard client transfer protocol (NNTP) allows unmodified clients access to this storage at each site.

This approach obviates the need for articles to be replicated to all participating servers. Instead of storing article data at each server, UsenetDHT stores article data on a small number of nodes in the DHT. The DHT distributes articles geographically, and can locate them quickly. It ensures high durability by re-replicating as necessary, maintaining the benefits of full replication without incurring the bandwidth and storage costs.

Using this approach, in an N-host system, each host will be able to offer a Usenet feed while only storing $O(1/N)$ as much data as it would have had to using a traditional news server. Instead of using local storage to hold a large number of articles for a short period of time, UsenetDHT allows each server to hold fewer articles but retain them for longer periods of time.

The bandwidth required to host a Usenet feed using UsenetDHT is proportional to the percentage of articles that are read rather than to the total number of articles posted. As

explained in Section 1.1, readers at sites for which UsenetDHT is targeted generally view only a small fraction of available content. Thus, we expect this design to translate into a significant reduction in the bandwidth required to "host" a full feed.

5.1 Usenet

The Usenet electronic bulletin board has been operational since 1981. Servers are distributed world-wide and traditionally serve readers located in the same administrative realm as the server. When each server enters, the operator arranges to peer with one or more nodes already in the network. Peering servers agree to exchange all new articles that they receive locally.

Over the years its use and implementation has evolved. Now people use Usenet in two primary ways. First, users continue to participate in discussions about specific topics, which are organized in newsgroups. The amount of traffic in these text groups has been relatively stable over the past years. Second, users post binary articles (encoded versions of pictures, audio files, and movies). This traffic is increasing rapidly, because Usenet provides an efficient way for users to distribute large multi-media files. Users can upload the article to Usenet once and then Usenet takes charge of distributing the article.

Usenet is unique in that it functions as a client-supported content distribution network. Usenet allows a content publisher to simply post any content and not worry about the distribution costs; all content is uniformly replicated to all interested servers. Servers control the volume of their feed by selecting groups they are interested in carrying. Commercial Usenet providers charge clients for access to content and can invest that money to provide bandwidth for peering and reading as well as local storage for high retention periods. In contrast, most CDNs are provided and paid for by content providers—Akamai, for example, makes money by charging content providers to mirror data.

Organization. Articles are organized into a hierarchy of newsgroups. Upon receiving each article, each peer determines which newsgroups the article is in (based on metadata included in the article) and updates an index for the group. The group indices are sent to news reading software and is used to summarize and organize displays of messages. The generation of summary data locally at each server is a natural consequence of Usenet's replication model and gives server administrators the flexibility to apply different retention or filtering policies.

Certain newsgroups are moderated to keep discussions on-topic and spam-free. Moderation is enforced by requiring a special header—articles posted without this header are sent to a pre-configured moderation address instead of being flood-filled as normal. Most servers today will also filter incoming and outgoing articles with CleanFeed [55] to remove articles that are considered to be spam.

Distribution. Usenet distributes articles using an overlay network of servers that are connected in a peer-to-peer topology. Servers are distributed world-wide and each server peers with its neighbors to replicate all articles that are posted to Usenet. The servers employ a

flood-fill algorithm using the NetNews Transfer Protocol (NNTP) to ensure that all articles reach all parties [4, 37].

Each Usenet site administrator has complete control over the other sites that it peers with, what groups it is interested in carrying and the particular articles in those groups that it chooses to keep. This flexibility is important as it allows administrators to utilize local storage capacity in a manner that best suits the needs of the site. For example, some Usenet providers choose to receive only articles affiliated with text newsgroups in order to minimize the bandwidth and storage required.

As a server receives new articles (either from local posters or its neighbors), it floods NNTP `CHECK` messages to all its other peers who have expressed interest in the newsgroup containing the article. If the remote peer does not have the message, the server feeds the new article to the peer with the `TAKETHIS` message. Because relationships are long-lived, one peer may batch articles for another when the other server is unavailable, but today's servers typically stream articles to peers in real-time.

Special messages called *control* messages are distributed through the regular distribution channels but have special meaning to servers. Control messages can create or remove newsgroups, and cancel news postings (i.e. remove them from local indices and storage).

Client read protocol. Servers give each new article it receives a per-newsgroup monotonically increasing number. To each client, it indicates the upper and lower bound of the article numbers available in the group. These counters are server-specific and will depend on the number of articles that this particular server has seen for a particular newsgroup. Each client keeps track of which article numbers it has read.

Clients selects a group, retrieves information about the new news, and then can request articles by number and by message-ID.

5.2 Architecture

In this section we present the high-level design of UsenetDHT, trace through the life of an article after it is posted, and discuss some trade-offs of our design.

5.2.1 Design

The main costs in today's Usenet come from the data transfer bandwidth required to replicate articles to all interested servers, and the disk storage requirements each server must provide to hold the replicated articles. UsenetDHT seeks to reduce these costs without affecting other aspects of Usenet.

The main goal of this system is to reduce the resources consumed by Usenet servers—specifically, the data transfer bandwidth and the on-disk storage requirements for articles—while preserving the major features of Usenet. UsenetDHT accomplishes this goal by replacing the local article storage at each server with shared storage provided by a DHT. This

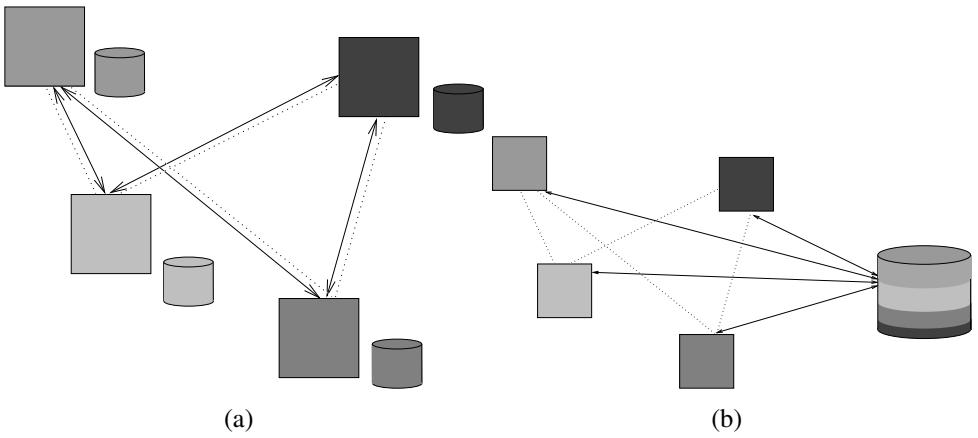


Figure 5-1: The design of Usenet (a) and UsenetDHT (b). Usenet servers are connected to form a mesh network and exchange article data (solid lines) and metadata such as article announcements and cancellations (dotted lines). Each Usenet server has enough local storage for all articles and metadata. UsenetDHT servers are connected by the same mesh network to exchange metadata, which is still stored locally. Articles are written to a large, shared DHT (at right), with backing storage contributed by the peer servers. The colors associate each participant with the storage they provide.

approach saves storage since articles are no longer massively replicated; it also saves bandwidth since servers only download articles that their clients actually read. Figure 5-1 illustrates this design. Dotted-lines in the figure denote the transfer of metadata, and solid lines denote article bodies.

Each article posted to Usenet has metadata—header information such as the subject, author, and newsgroups—in addition to the article itself. Articles entering a UsenetDHT deployment (for example, from a traditional Usenet feed or a local user) will come with metadata and the article bundled together. UsenetDHT floods the metadata among its participating peers in the same way as Usenet does. UsenetDHT, however, stores the articles in a DHT.

In UsenetDHT, each site contributes one or more servers with dedicated storage to form a DHT, which acts like a virtual shared disk. The DHT relieves UsenetDHT from solving the problem of providing robust storage; DHT algorithms deal with problems of data placement, maintenance in the face of failures, and load balance as the number of servers and objects in the system increases [13, 78]. To support a full feed, each server in a homogeneous deployment need provide only $O(1/n)$-th of the storage it would need to support a full feed by itself.

NNTP front-ends store incoming articles into the DHT using `put` calls; these articles may come from local users or from feeds external to the UsenetDHT deployment. To send articles upstream to the larger Usenet, front-ends in a UsenetDHT deployment have the option of arranging a direct peering relationship with an external peer or designating a single front-end to handle external connectivity.

Usenet is organized into newsgroups; when an article is posted, it includes metadata in its

headers that tells the NNTP front-ends which groups should hold the article. In UsenetDHT, each NNTP front-end receives a copy of all headers and uses that information to build up a mapping of newsgroups to articles stored to local disk for presentation to its local clients. All indexing of articles remains local to servers. In particular, each front-end keeps an article index independently from other sites. UsenetDHT does not store group lists or the mapping from newsgroups to articles in the DHT.

Distributing metadata to all sites has several advantages. First, it guarantees that a site will be able to respond to NNTP commands such as `LIST` and `XOVER` without consulting other front-ends. These commands are used by client software to construct and filter lists of articles to present to actual users, before any articles are downloaded and read. Second, it leaves sites in control over the contents of a group as presented to their local users. In particular, it allows sites to have different policies for filtering spam, handling moderation, and processing cancellations.

UsenetDHT nodes exchange metadata and control-data using the same flood-fill techniques as Usenet. The flooding network used by UsenetDHT to propagate metadata follows peering relationships established by the system administrators. However, the article bodies are elided from the news feed between servers. This approach allows cancellation and moderation of articles to work much as they do now; for example, cancellation simply updates the local group index to exclude the canceled article and no DHT storage needs to be touched.

Clients access UsenetDHT through the NNTP front-ends. When a user reads an article, the NNTP front-end retrieves the article using a DHT `get` call and caches it. Local caching is required in order to reduce load on other DHT servers in the system and also to ensure that UsenetDHT never sends more traffic than a regular Usenet mesh feed would. If sites cache locally, no DHT server is likely to experience more than N remote read requests for the DHT object for a given article. This cache can also be shared between servers at a site. Each site will need to determine an appropriate cache size and eviction policy that will allow it to serve its readership efficiently.

5.2.2 Write and read walk-through

To demonstrate the flow of articles in UsenetDHT more precisely, this section traces the posting and reading of an article. Figure 5-2 summarizes this process.

A news reader posts an article using the standard NNTP protocol, contacting a local NNTP front-end. The reader is unaware that the front-end is part of UsenetDHT, and not a standard Usenet server. Upon receiving the posting, the UsenetDHT front-end uses `put` to insert the article in the DHT. In the `put` call, the front-end uses the SHA-1 hash of the article's content as the key: since DHash partitions data across servers by the key, using a hash function ensures articles will be distributed evenly across the participating servers. By providing this key in a `get` call, any UsenetDHT front-end can retrieve the article from the DHT. The use of a content-hash key also allows front-ends to detect data corruption by verifying the integrity of data received over the network.

After the article has been successfully stored in the DHT, the front-end propagates the article's metadata to its peers using a `TAKEDHT` NNTP message. This message is similar to

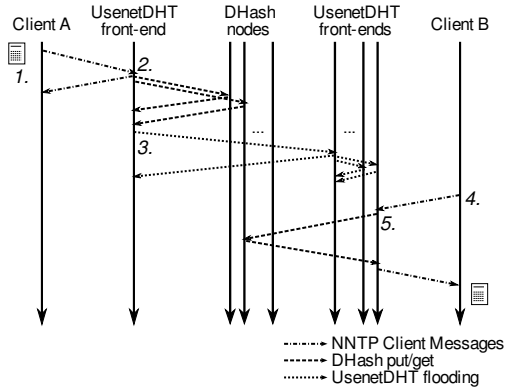


Figure 5-2: Messages exchanged during UsenetDHT reads and writes. 1. Client A posts an article to his local NNTP front-end. 2. The front-end stores the article in DHash (via a DHash gateway, not shown). 3. After successful writes, the front-end propagates the article's metadata to other front-ends. 4. Client B checks for new news and asks for the article. 5. Client B's front-end retrieves the article from the DHT and returns it to her.

the standard `TAKETHIS` message but only includes the header information and the content-hash key (as an `X-ChordID` header). This information is sufficient for the peers to insert the article into their local group indices, provide a summary of the article to readers connecting to the front-end and retrieve the contents of the article when a reader requests it. Each front-end, upon receiving the article, also shares the announcement with its other peers. In this manner, the article's existence is eventually flooded to all front-ends in the deployment.

When a user wishes to read a newsgroup, his news reader software requests a list of new articles from his UsenetDHT front-end. The front-end responds with a summary of articles that it has accumulated from its peers. This summary is used by the reader to construct a view of the newsgroup. When the client requests an article body, the front-end first checks its local cache; if the article is not present, it calls `get`, supplying the key for the article as argument. When the front-end obtains the article data from the DHT, it inserts the article into the cache and returns the article to the reader. As with posting, the reader is unaware that the news server is part of UsenetDHT.

5.2.3 Expiration

UsenetDHT will insert dozens of objects into the DHT per second, resulting in millions of objects per day. After an initial start-up period, the DHT will be operating at full storage capacity. Thus, some mechanism is needed to delete older objects to make room for newer ones.

Usenet servers have long used *expiration times* to bound the lifetime of articles. Each deployment of UsenetDHT sets a single common expiration policy across all participants. This policy can vary according to the type of newsgroup (e.g., preserving text discussions for

longer than binary articles). A common policy is required to ensure that the list of articles for a given group at any UsenetDHT front-end will accurately reflect the articles that are available in the underlying DHT.

5.2.4 Trade-offs

UsenetDHT converts a fully decentralized system based on pushing content to all servers into a partially decentralized one, where individual front-ends pull the content from their peers. Thus, servers must participate in processing DHT lookups for all articles, even for readers at other sites. Conversely, each server depends on other servers to respond to its read requests. This motivates requiring a trusted set of participating sites.

Global configuration. The primary drawback of UsenetDHT is a loss of control by individual administrators over what data (articles) are stored on and transit their machines. In a DHT-based storage system, article data will be evenly spread across all participating nodes. From a resource standpoint, this drawback is probably not critical: each server in UsenetDHT need only provide a fraction of the space required to store a feed. Even if a traditional server only served a fraction of newsgroups, its storage requirements are not likely to rise if it participates in UsenetDHT. In addition, sites can run virtual nodes to match DHT storage requirements with available capacity: each virtual node allows a physical node to claim responsibility for an additional fraction of the objects stored in the DHT.

The global nature of storage also means that all nodes must participate in processing DHT lookups for articles, even those in groups that they do not wish to carry. Similarly, article expiration times must also be handled on a system-wide level. Objects associated with articles can only be removed when the article has expired from the indices of all servers. This global parameter is set out-of-band.

Filtering. One common policy that UsenetDHT would like to support is the ability for servers to decide not to carry particular newsgroups, such as adult newsgroups. Normally, such servers would request not to receive articles associated with unwanted groups from their upstream server. In UsenetDHT, a fraction of the articles from *all* groups will be randomly distributed to all servers. One possible solution may be to apply techniques that divide a Chord ring into subrings such that a lookup could be targeted at an item's successor in a given subset of nodes [39]. Subrings could be used so that storage for content such as adult material would be limited to a single subring: lookups for articles in adult groups would be confined within the subring.

A different challenge is effectively filtering spam; the previous solution would not work since spam appears in all newsgroups. In total, the use of UsenetDHT should reduce the storage impact of spam since each spam message will only be stored once. However, administrators will still want to reduce the presence of spam in newsgroup indices—naïvely, any node wishing to do this would have to retrieve every complete article to determine if it is spam. This approach would lose all the bandwidth benefits of UsenetDHT. The most

efficient method for dealing with spam is to apply ingress filtering: as Usenet articles enter a UsenetDHT system (or are originated from members of the UsenetDHT deployment), they should be subjected to spam screening. Also, existing techniques such as spam cancellation can still work in UsenetDHT—a small number of sites determine what messages are spam (or are otherwise undesirable) and publish cancel messages. Other sites process the cancel messages to filter out spam locally. Sites would continue to filter local posts to prevent spam from originating locally as well.

Read latency. The read latency of the DHT will be visible to end-users when they request an uncached article: reading a small text article requires the DHT to perform a single object download. The time required by the DHT to perform the fetch is likely to be significantly higher than the time to perform a similar operation on a local server. This delay can be partially masked by pre-fetching articles in a newsgroup when the indexing information is requested but before they are actually read. We believe that the bandwidth and storage savings of UsenetDHT are worth the delay imposed by the DHT.

Flooding overlay. The metadata distribution framework may be inefficient and cause metadata to reach the same site multiple times. A reasonable alternative may be to construct a distribution tree automatically [35]. An efficient broadcast tree would reduce link stress by ensuring that data is transmitted over each link only once. However, the metadata streams are relatively small and the design avoids the complexity and possible fragility involved in deploying such an overlay.

5.3 Anticipated savings

In contrast to more traditional file sharing systems like Gnutella or Kazaa (studied in [5]), nodes that serve a system like UsenetDHT will show a significantly smaller degree of membership churn, such as those described in Table 3-2. The nodes used as DHT nodes for UsenetDHT will be similar to those that are currently used for Usenet servers today: high-performance, well-connected machines stored in data centers. Thus there will be high capacity between cluster internal-nodes and wide-area links between nodes at different sites. A stable membership makes it easier for the DHT to provide highly reliable storage [7].

In the rest of this section, we quantify the potential bandwidth and storage requirements of a UsenetDHT server compared to a Usenet server based on available statistics about the current Usenet. We also describe what performance we require from the underlying DHT.

5.3.1 Analysis model

The resource usage of a UsenetDHT server depends on the behavior of readers; we parametrize our analysis based on a simplistic model of the input rate to the system and estimated readership. Let N be the number of servers in the system. Let a represent the average number

	Total Bytes Transferred	Storage
Usenet	$2\overline{b}$	$\overline{b}$
UsenetDHT	$2\overline{b}r + 2 \cdot 1024\overline{a}$	$2\overline{b}/N + 1024\overline{a}$

Table 5-1: UsenetDHT reduces the bandwidth and storage requirements of hosting a Usenet feed in proportion to the fraction of articles read (r) and the number of servers in the network (N), respectively. This table compares the current transfer and storage requirements per day for a full Usenet feed in both systems, where $\overline{b}$ represents the total number of bytes for articles injected each day. A single peer is assumed.

of articles injected into the system per second. Correspondingly, let b represent the average number of bytes injected per second.

Our actual deployment is based on the Usenet feed at CSAIL, which is ranked approximately 600th out of the Top1000 Usenet servers world-wide [24]. The CSAIL feed has an average accept rate around $a = 13$ articles per second, seeing approximately 1 million articles per day. It sees traffic at approximately $b = 2.5$ Mbyte/s. This traffic is at least 75% binary traffic. Binary articles have a median article size of 240 Kbyte; text articles are two orders of magnitude smaller, with a median size of 2.4 Kbyte.

Statistics from other news servers (e.g., `spool-1t2.cs.clubint.net`) show that to assemble more complete feeds, sites can experience much higher volumes, with an aggregate $a \approx 115$ articles per second and $b \approx 28$ Mbyte per second [23]. When we examine the storage requirements of a UsenetDHT server it will be convenient to consider data stored per day: we will write $\overline{a}$ for the total number of articles injected on average each day ($\overline{a} = 86400a$) and $\overline{b}$ for total bytes injected per day.

To model the readership patterns of Usenet, we introduce r, the average percentage of unique articles read per site. Unfortunately, few studies have been done to measure how many articles are actually read on different servers. In 1998, Saito *et al.* observed that roughly 36% of all incoming articles were read on the server at Compaq [71]. Fewer than 1% of articles at the MIT and CSAIL news servers are read [68, 88]. Today, because of large traffic volume, many networks outsource their news service to large, well-connected providers, such as GigaNews. Byte-transfer statistics for the month of January 2004 from a small ISP that outsources news service to GigaNews suggest that the ISP's approximately 5000 customers read approximately 1% of the total monthly news. In fact, the trends from that ISP show that the number of bytes downloaded has remained relatively constant around 300 Gbyte per month over the past year. This may mean that r will decrease over time if the growth of Usenet remains exponential.

For the purposes of this analysis, we will treat these parameters as constants, though of course, they will change over time.

5.3.2 Storage

UsenetDHT reduces system-wide storage by storing articles once in the DHT instead of copying them to each site. The storage requirements of a node are proportional to the amount of data in the system and the replication overhead of the underlying DHT and inversely proportional to the number of participating servers.

Since UsenetDHT will be deployed on comparatively stable machines, we set it to use DHash with a replication factor of 2. This means that the system receives $2b$ bytes of new article data each second. This load is spread out over all N servers in the system instead of being replicated at all hosts, resulting in an overall per-host load that is $2/N$ times the load of traditional Usenet. If the number of servers participating in UsenetDHT increases and the number of articles posted remains constant, each server must bear less load. Because each server must dedicate a factor of N less storage, UsenetDHT should allow articles to be retained longer within the system.

There is also a small incremental cost required for local indexing. Suppose that each article requires about 1 Kbytes to store the overview data, which includes article author, subject, message-id, date, and references headers. This data adds an additional $1024\overline{a}$ bytes per day to the cost of supporting a full feed. Given the average size of each article, this is less than 1% of the total data required for storing the articles themselves.

Sites must also provide some storage space for caching locally read articles. The sizing and effectiveness of this cache is dependent on the size of the reader population and the diversity of articles that they retrieve.

The total daily storage requirement for a UsenetDHT server is $2\overline{b}/N + C + 1024\overline{a}$ where C is the size of the server's article cache. This cost differs from the cost of a traditional Usenet server by roughly a factor of $2/N$. A Usenet server requires $\overline{b} + 1024\overline{a}$ bytes of storage daily.

5.3.3 Bandwidth

When comparing UsenetDHT bandwidth requirements to Usenet, we will not consider the cost of sending articles to readers for Usenet servers since these costs remain the same regardless of the server type and, further, readers are often on the same network as the server where bandwidth is cheaper. A Usenet server's wide-area bandwidth requirements are equal simply to the size of the feed it serves. In our notation, a Usenet server is required to read b bytes per second from the network.

A UsenetDHT server has downstream bandwidth requirements for receiving metadata, storing articles, and serving reads. The header and DHT information for each article corresponds to $1024a$ bytes per second (assuming 1 Kbyte per article). This header overhead may need to be multiplied by a small factor, corresponding to the overhead of communicating with multiple peers. Incoming article bodies arrive at a rate of $2b/N$, corresponding to a node's fraction of DHT. Finally, servers must download articles from the DHT to satisfy requests by local readers. If the average actual fraction of read articles is r, each server will require roughly rb bytes per second of additional downstream bandwidth. This total is $1024a + (2/N + r)b$ bytes per second.

Upstream bandwidth is required for propagating any articles generated locally (unaffected by UsenetDHT), and to satisfy reads from other nodes in the system. Each server requests *rb* bytes per second from the DHT. This load will be spread out over all *N* servers. Correspondingly, each participating server must send rb/N bytes per second to satisfy read requests from each other server in the system. Thus, in aggregate, each site must have *rb* bytes per second of upstream bandwidth.

Based on the upstream and downstream requirements, the total bytes transferred per day will be $1024\overline{a} + 2(\frac{1}{N} + r)\overline{b}$. Thus, a full-feed deployment with 100 nodes and $r = 0.01$ in 2007 would require that each DHT node provide 2 Mbyte/s of symmetric bandwidth, as opposed to 50 Mbyte/s. Benchmarks in the next Chapter verify that DHash is able to meet the performance requirements of the system.

5.4 Implementation

UsenetDHT is a basic NNTP implementation in 3,000 lines of C++ using the `libasync` libraries. Our server has support for receiving and forwarding news feeds, inserting articles into the DHT, and handling read/post requests from clients.

Our UsenetDHT implementation stores each article as a single DHash content-hash object; use of a single object minimizes the overhead incurred by DHash during storage as compared to chunking articles into, for example, 8 Kbyte blocks [16]. Each server maintains a local index for each group that describes the Message-IDs of each article in the group and its corresponding content-hash. The content-hash is included as an `X-ChordID` header that is included in the metadata only feed.

For performance, the UsenetDHT implementation draws largely on techniques used in existing open-source Usenet servers. For example, UsenetDHT stores overview data in a BerkeleyDB database, similar to INN's overview database (`ovdb`). An in-memory history cache is used to remember what articles have been recently received. This ensures that checks for duplicate articles do not need to go to disk. The current UsenetDHT implementation does not yet support a DHT read-cache.

One significant problem our server faces is that of performing history checks. The current implementation must perform a seek at each node before agreeing to accept a new article; this seek checks whether the existing article is known to the node already, compared to all other articles in the system. Most Usenet servers use a heuristic of keeping an in-memory (but disk-backed) list of recent articles that can be efficiently searched; this allows for the possibility of adding an article to a group more than once but in the absence of slow batching systems that deliver articles long after they are originally inserted, this possibility is likely to be remote.

Chapter 6

Evaluation

In this chapter, we demonstrate that:

- Our test deployment of UsenetDHT is able to support a live Usenet feed;
- Passing Tone identifies repairs following transient failures, permanent failures and server additions without interfering with normal operations; and
- Our implementation of DHash and UsenetDHT scales with the available resources.

These tests with live data validate the simulation results shown in Chapter 3.

6.1 Evaluation method

Passing Tone is evaluated under load using a live wide-area deployment. The deployment consists of twelve machines at universities in the United States: four machines are located at MIT, two at NYU, and one each at the University of Massachusetts (Amherst), the University of Michigan, Carnegie Mellon University, the University of California (San Diego), and the University of Washington. Access to these machines was provided by colleagues at these institutions and by the RON test-bed. While these machines are deployed in the wide area, they are relatively well connected, many of them via Internet2.

These machines are lightly loaded, stable and have high disk capacity. Each machine participating in the deployment has at least a single 2.4 Ghz CPU, 1 Gbyte of RAM and UDMA133 SATA disks with at least 120 Gbyte free. These machines are not directly under our control and are shared with other users; the write caching provided by the disks themselves is enabled on these machines. This may lead to higher write throughput, but synchronous writes requested by the DHash or UsenetDHT implementation (to ensure durability) may return without having actually been written to the disk.

The load for the live deployment is a mirror of the CSAIL news feed; the CSAIL feed is ranked 619$^{\text{th}}$ in the Usenet Top 1000 servers [24]. This feed sees one million articles per day on average and is around 10% of the full feed. This feed generates a roughly continuous 2.5 Mbyte/s of write traffic. Binary articles have a median article size of 240 Kbyte; text

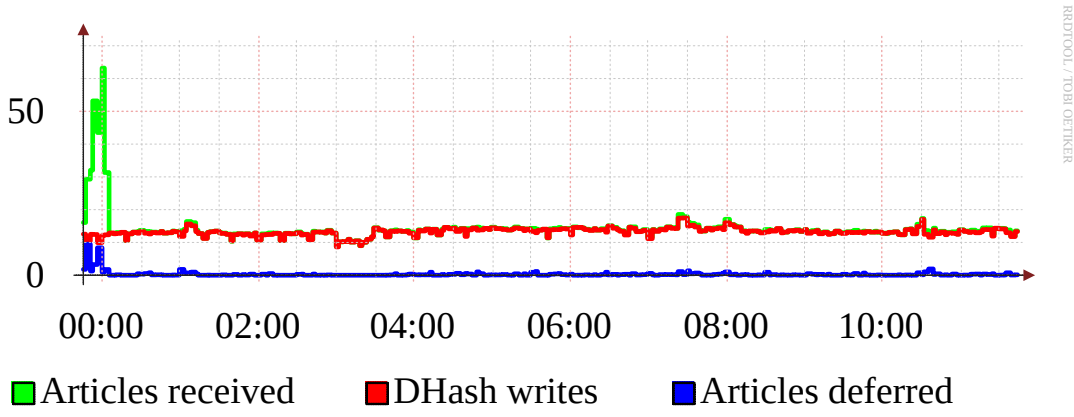


Figure 6-1: Articles received, posted and deferred by the CSAIL UsenetDHT deployment over a twelve hour period. During normal operation, deferrals are very rare.

articles are two orders of magnitude smaller, with a median size of 2.4 Kbyte. The overall average article size is approximately 173 Kbyte.

Microbenchmarks, run on a private gigabit switched local area network, demonstrate the scalability potential of DHash and UsenetDHT. Using a local network eliminates network bottlenecks and focuses on the disk and memory performance.

6.2 UsenetDHT wide-area performance

The CSAIL Usenet feed receives on average 14 articles per second. This section demonstrates that UsenetDHT, with DHash and Passing Tone, is able to meet the goal of supporting CSAIL's Usenet feed. DHash is configured to replicate articles twice, prior to any maintenance; thus to support the CSAIL feed, the system must write at least 28 replicas per second. Our deployment has supported this feed since July 2007. This section also demonstrates that our implementation supports distributed reads at an aggregate 30.6 Mbyte/s.

Figure 6-1 shows the number of articles received by UsenetDHT, posted into DHash writes, and deferred. Deferrals occur when DHash's write window is full—during initial server startup, deferrals occur since there is a backlog of posts from the upstream feed that arrive faster than the local DHash server can write replicas to remote sites. The upstream later re-sends deferred posts. During normal operation, deferrals do not occur, showing that UsenetDHT and DHash can keep up with the normal workload from the CSAIL feed.

To evaluate the read capability of the system, we use clients distributed on the PlanetLab test bed. We selected 100 of the machines with the lowest load that were geographically close to one of the seven sites that were running at the time of the experiment using CoMon [56]. Each client machine ran five parallel NNTP clients to the UsenetDHT front-end closest to them and downloaded articles continuously from newsgroups chosen at random.

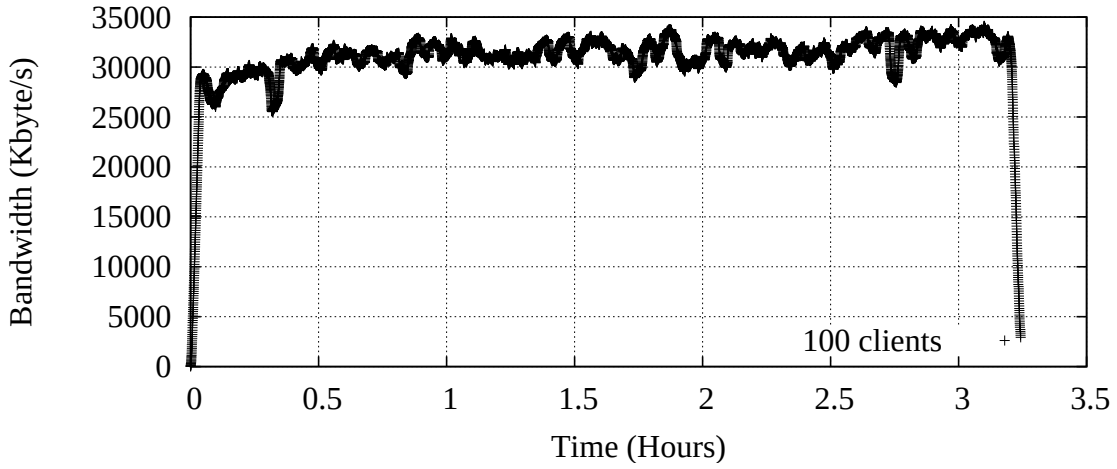


Figure 6-2: UsenetDHT aggregate read throughput. 100 hosts started article reads continuously against 12 servers.

Figure 6-2 plots the achieved aggregate bandwidth over time: the client machines were able to achieve an collectively 30.6 Mbyte/s or approximately 313 Kbyte/s per client machine. This corresponds to reading on median 2612 Kbyte/s per disk. We estimate that each article read requires seven disk seeks including metadata lookup at the Usenet server, object offset lookup, opening the relevant file and its inode, indirect and double-indirect blocks, data read, and atime update. With at least 70ms of total seek time per access, this means that each disk can support no more than 14 reads per second: given an average article size of 173 Kbyte ($14 \times 173 = 2422$ Kbyte/s), this corresponds well with the observed throughput per disk.

6.3 Passing Tone efficiency

When failures do arise, Passing Tone is able to identify repairs without substantially affecting the ability of the system to process writes. Figure 6-3 demonstrates the behavior of Passing Tone under transient failures and server addition in a six hour window, by showing the total number of objects written: the number of objects repaired is shown in a separate color over the number of objects written. At 18:35, we simulated a 15 minute crash-and-reboot cycle on a server responsible for 8.5% of the data in the system. Passing Tone immediately begins repairing and ceases when that server comes back online at 18:50. A more permanent failure would involve a longer period of repairs, as repair speed is limited largely by bandwidth.

To demonstrate this limit, the next event occurs at 20:15, where we add a new server, responsible for 3% of the data in the system. It begins transferring objects from its successor and moves 11 Gbyte of data in just over 2.4 hours. This corresponds to 1.2 Mbyte/s, which is what the link between that server and its successor can support.

Finally, we demonstrate a short transient failure of the same new server at 00:05. Its

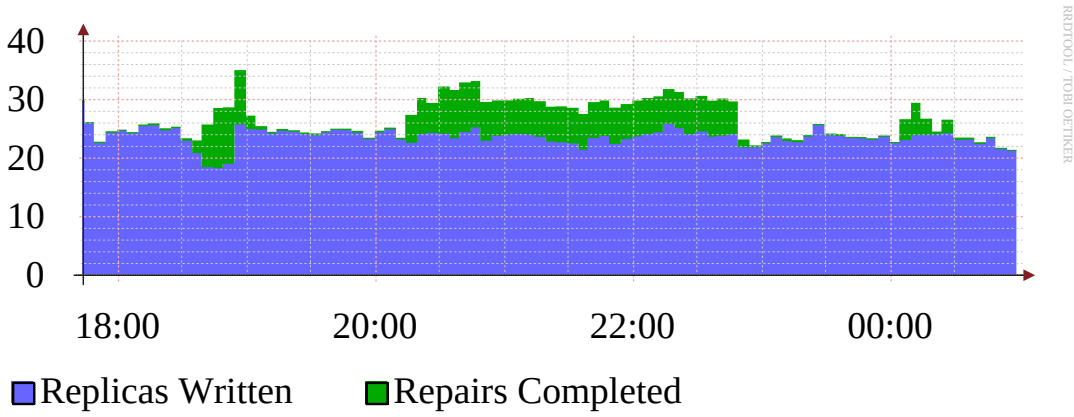


Figure 6-3: Number of repairs and number of writes over time, where we introduce a simple transient failure, a server addition, and a second transient failure. The second failure demonstrates behavior under recovery and also what would have happened in a permanent failure.

neighbors begin repairing the objects that were inserted to the failed server during its four hour lifetime: however, none of the objects that were stored on the original successor needed to be repaired because its replicas were not deleted. When the new server is brought back online at 00:20, there are a few transient repairs to bring it up to date with the objects that were written during its downtime but not a repeat of the four hour initial transfer. This second transient failure also indicates how Passing Tone would behave after a permanent failure: we expect the transient failures will create a small buffer of extra replicas so that when a permanent failure does occur, only those (relatively) few objects that have been inserted since will require re-replication.

6.4 DHash microbenchmarks

The microbenchmarks were conducted on a cluster of 8 Dell PowerEdge SC1425 servers, with Intel® Xeon™ 2.80 Ghz CPUs (HyperThreading disabled), 1 Gbyte of RAM, and dual Maxtor SATA disks. Each machine runs FreeBSD 5.4p22. The machines are interconnected with a dedicated Gigabit Ethernet switch. These machines are under our direct control and IDE write caching is disabled (with `hw.ata.wc="0"` set in `/boot/loader.conf`). They can do sustained writes at 7 Mbyte/s on average and read at 40 Mbyte/s; a half stroke seek takes approximately 14ms. A separate set of machines are used for generating client load.

To test the read and write scalability of the implementation, we configure one client per DHT server and direct each client to write (and read) a 2 Gbyte synthetic stream of 200 Kbyte objects as fast as possible. Replication, maintenance and expiration are disabled. The resulting

	Median bandwidth (Kbyte/s)	
Number of DHT nodes	Write	Read
1	6600	14000
2	12400	19000
3	18800	28000
4	24400	33900
5	30600	42600
6	36200	49200
7	42600	57200
8	46200	63300

Table 6-1: Read and write microbenchmarks

load is spread out over all the servers in the system. The results are shown in Table 6-1.

Each individual machine contributes on average an additional 5.7 Mbyte/s of write throughput; thus in an environment that is not network constrained, our implementation easily operates each server at close to its disk bottleneck. For reads, each additional machine contributes on average 7 Mbyte/s of read throughput. This result is largely driven by seeks. In the worst case, each 200 Kbyte object requires at least one seek to look up the object's offset in the metadata database, one seek to read the object and possibly one seek to update the atime on the inode. In this case, it would require 40ms simply to seek, limiting the number of objects read per disk to 25 per second (or $\approx$ 5 Mbyte/s). Systems with fewer servers will do better on this particular benchmark as objects are read in order of insertion. Thus, with fewer write workloads intermingled, fewer seeks will be required and operating system read-ahead may work well. The observed 14 Mbyte/s in the single server case corresponds to one seek per read on average, where it is likely that BerkeleyDB has cached the offset pages and the OS read-ahead is successful. In the multi-server cases, the 7 Mbyte/s average increase corresponds well to two seeks per read.

Thus, in a well-provisioned local network, the DHash implementation can write a 2.5 Mbyte/s Usenet feed to a single server (though with extremely limited retention, since standard disks available today would fill in hours or days). By adding additional servers (and hence disk arms and disk capacity), DHash is also able to scale as needed to support reader traffic and increase retention.

Chapter 7

Related work

Passing Tone's approach to providing durability in distributed hash tables builds on ideas and techniques seen in many previous distributed systems. Compared to these earlier systems, DHash and Passing Tone target reliable, high-performance storage in the wide-area, extending prior work by Dabek *et al.* [13, 14, 17]. Similarly, many ideas have been proposed and discussed for reducing the bandwidth costs of Usenet. This chapter first discusses related systems and algorithms for DHT-based storage and second alternate systems aimed at reducing Usenet's bandwidth usage.

7.1 DHT and maintenance

Scaling peer-to-peer storage. Early work on scaling storage systems using peer-to-peer techniques has ranged from LH* [48] to those built on DHTs such as PAST [69] and Oceanstore/Pond [65]. The limit to which such wide-area peer-to-peer systems can scale was discussed by Blake and Rodrigues, who observed that wide-area storage systems built on unreliable nodes cannot store a large amount of data [7]. Their analysis is based on the amount of data that a host can copy during its lifetime and mirrors our discussion of feasibility. The main difference in our work is that we focus on systems with stable membership where data loss is driven solely by disk failure, while they assumed a system with continual membership turnover.

Distributed storage systems. Other researchers and developers have built systems that also focus on node populations that are not based on peer-to-peer file sharing applications.

The most similar application to DHash is OpenDHT, which provides a DHT for public research use and is deployed on PlanetLab [63, 66]. Compared to DHash, OpenDHT focuses on storing small data objects and providing fairness across multiple applications. OpenDHT does not need to consider moving large objects across nodes. OpenDHT uses Merkle synchronization trees in maintenance as well but constructs them dynamically, based on insertion time, for each neighbor that is a synchronization partner. Passing Tone's division of local and global maintenance (originally proposed by Cates [12]) has also been used in OpenDHT.

The Glacier distributed storage system takes redundancy to the extreme [33]. Glacier uses massive replication to provide data durability across large-scale correlated failure events. The resulting trade-offs are quite different from those of Passing Tone, which is designed to handle a continuous stream of small-scale failure events. For example, due to its high replication level, Glacier can afford very long timeouts and thus mask almost all transient failures. Glacier uses erasure coding and stores an order of magnitude more data than is inserted in order to survive the failure of over 60% of the nodes in the system.

Antiquity, a storage system that uses a distributed hash table as a data location service, resists Byzantine failures by using secure, verifiable logs [86]. Antiquity provides an extent-storage interface, as opposed to the simple *put*/*get* interface provided by DHash. While Antiquity has been tested at scale, the largest reported deployment has stored only 84 GByte of data; because of its tolerance to Byzantine faults, cluster throughput for Antiquity is under 1 Mbyte/s.

A major commercial application that applies similar techniques is Amazon's Dynamo system. Dynamo is based on DHT ideas and supports numerous Amazon functions, including maintaining shopping cart state [19]. Dynamo's techniques for load balance, performance and flexibility are highly relevant to DHash. Unlike DHash, Dynamo is deployed in more cluster-oriented environments, with few wide-area links. Dynamo has a complete implementation of mutable data support. Dynamo's permanent failure recovery is manually controlled, where disk failures are explicitly verified by an administrator before catastrophic recovery occurs. All system detected failures are considered temporary and objects are buffered on backup nodes with the explicit intent of returning them to the temporarily failed node upon its return. Merkle synchronization trees are again used to identify differences between objects on different nodes.

The Coral and CobWeb CDNs provide high-bandwidth content distribution using DHT ideas [21, 76]. Coral uses distributed sloppy hash tables that optimize for locality and CobWeb locates data using Pastry [10] (with content optimally replicated using Beehive [62]). Both Coral and CobWeb have been successfully deployed and supports extremely high-bandwidth usage. However, CDNs cache content with the aim of reducing latency and absorbing load from an origin server. For example, Coral is a caching system, so it does not have any requirements for data maintenance. UsenetDHT operates in a system without origin servers and must guarantee durability of objects stored.

Moving away from distributed hash tables, SafeStore provides an architecture for managing reliable storage [42]. Like Glacier, it encourages the use of erasure code. Unlike Passing Tone, Glacier and OpenDHT, SafeStore makes explicit distinctions between different storage providers (SSPs) to ensure diversity—storage providers are located in different geographic locations, are operated by different organizations, and may even run different software implementations. SSPs are untrusted and periodically audited to determine data-loss; unlike DHTs where each node takes responsibility for auditing specific ranges, SafeStore puts the burden of auditing on the owner/writer of the data.

Replication. Replication has been widely used to reduce the risk of data loss and increase data availability in storage systems (e.g., RAID [57], System R duplex disks [29], Harp [47], xFS [3], Petal [43], DDS [30], GFS [25]). The algorithms traditionally used to create and maintain data redundancy are tailored for the environment in which these systems operate: well-connected hosts that rarely lose data or become unavailable. This thesis focuses on wide-area systems that are somewhat bandwidth-limited, where transient network failures are common, and where it is difficult to tell the difference between transient failures and disk failures.

The selection of a target replication level for surviving bursts differs from many traditional fault tolerant storage systems. Such systems, designed for single-site clusters, typically aim to continue operating despite some fixed number of failures and choose number of replicas so that a voting algorithm can ensure correct updates in the presence of partitions or Byzantine failures [11, 30, 47, 70].

FAB [70] and Chain Replication [84] both consider how the number of possible replicas sets affects data durability. The two come to opposite conclusions: FAB recommends a small number of replica sets since more replica sets provide more ways for data to fail; chain replication recommends many replica sets to increase repair parallelism and thus reduce repair time. These observations are both correct: choosing a replica placement strategy requires balancing the probability of losing some data item during a simultaneous failure (by limiting the number of replica sets) and improving the ability of the system to tolerate a higher average failure rate (by increasing the number of replica sets and reconstruction parallelism).

Synchronization. Nodes running Passing Tone or other maintenance algorithms must learn where objects are stored and whether a particular neighbor has a given object. This is done using set reconciliation protocols for synchronization: we have two sets S_A and S_B and wish to allow A to determine which elements are in $S_B \setminus S_A$ and B to determine $S_A \setminus S_B$. In the context of DHT replica maintenance, S_n contain the set of bitstrings corresponding to the keys of objects stored on node n.

In the simplest synchronization protocol, one side sends a complete list of objects to the other, which meets this goal but is obviously inefficient: repeated synchronizations result in the unnecessary duplication of information about each of the keys that have not changed. Glacier [33] employs a variant of this approach by exchanging Bloom filters: in essence, nodes exchange a sample of the keys and then compare these samples to detect differences, in an attempt to limit the frequency of full key exchange. Since Glacier rotates Bloom filters periodically, it achieves eventual consistency, just as Passing Tone does. However, using Bloom filters still repetitively exchanges information about objects and provides only a linear reduction in bandwidth that still grows with the number of objects as opposed to the number of differences.

An alternative to key list exchange is to remember some state about the previous synchronization. For example, nodes could maintain an append-only file to list all keys on a given node. Two nodes would synchronize by exchanging any information appended since the last known end-of-file marker; two nodes syncing for the first time would simply exchange the

whole list of keys. The primary benefit of this approach is its implementation simplicity.

There are several notable disadvantages, however. First, keys that are common to both nodes (i.e., obtained independently, perhaps inserted simultaneously by a writer) still must be exchanged as there is no mechanism for skipping common keys that were appended to both nodes. There is also no mechanism for efficiently doing range-limited synchronization; range-limited synchronization is normally used to limit the keys compared to those that are shared by the synchronizing node. Arbitrary ranges must be supported to deal with evolving partitions of responsibility as different nodes join and leave the system. Finally, the use of an end-of-file marker requires that deletions be explicitly included in the key database.

Despite a high number of objects, the Merkle synchronization protocol allow Passing Tone to identify missing replicas efficiently in network bandwidth and also in terms of state required per node, without the complexity of aggregation and Bloom filters as used in Glacier [33] or the need to aggregate replica information as in Carbonite [13]. The Merkle synchronization protocol used by Passing Tone requires up to $O(\log |S_A|)$ messages per difference [12]. The synchronization protocol supports limiting the reconciliation to a range-limited subset of keys, based on Chord identifiers, which is crucial for Passing Tone. Merkle trees, as used in Passing Tone, also allow Passing Tone to avoid any requirement that a node store state about data on other nodes.

Minsky *et al* have developed a synchronization protocol that is similar to Merkle but requires less communication in the base case [51], The structure of the algorithm is similar to Merkle trees: it divides the b-bit key space into p sub-partitions and only synchronizes those sub-partitions where there are differences. (In this parlance, Merkle has $b = 160$ with $p = 64$.) Where it wins is in its base case: instead of sending over a list of 64 keys, it uses a characteristic polynomial, interpolation and zero-finding to resolve up to $\bar{m}$ differences at a time. Implementing an on-disk storage for this protocol, including support for expiration and dynamic sub-ranges would probably have required at least as much work as was performed in this thesis—since Merkle synchronization represents only a small fraction of the bandwidth used by DHash and Passing Tone, we have not pursued implementing this improved algorithm.

Maintenance. We have explored several non-synchronization based approaches to maintenance. For example, in the Tempo algorithm, we propose continuously replicating objects to proactively handle failures [73]. Tempo was inspired by the Accordion routing protocol's use of a bandwidth cap for routing table maintenance: a similar cap for maintenance would avoid the problem of spikes in bandwidth usage caused by responding to failures. Tempo would try to provide the best durability possible within this bandwidth cap. However, users typically are more interested in absolute durability than achieving an unknown amount of durability with a constant amount of bandwidth usage. Tempo's need to count the number of available replicas (in order to prioritize objects for future repair) also requires coordination among all live replica holders which is difficult to implement correctly and efficiently.

Another alternate approach we considered was attempting to identify objects and ranges of objects for repair via sampling: in this approach, instead of synchronizing, a node will probe a neighbor to attempt to identify objects that are missing remotely. We were unable to

determine efficient algorithms for generating probes that allowed us to successfully find the missing object out of the thousands or tens of thousands of objects stored on a given node. The existing Merkle synchronization protocol was clearly more accurate, and results in fast identification of missing objects.

Passing Tone optimizes maintenance for accuracy and dealing with expiration. Like Carbonite, Passing Tone keeps extra copies of objects on nodes in the successor list to act as a buffer that protects the number of available objects from falling during transient failures [13].

7.2 Usenet

There have been some other proposals to reduce the resource consumption of Usenet. Newscaster [8] examined using IP multicast to transfer news articles to many Usenet servers at once. Each news article only has to travel over backbone links once, as long as no retransmissions are needed. In addition, news propagation times are reduced. However, Newscaster still requires that each Usenet server maintain its own local replica of all the newsgroups. Also, each news server still consumes the full news feed bandwidth across the network links closest to the server.

The Cyclic News Filesystem [22] is a file system that improves the speed at which news articles can be stored and expired. It uses a user-level circular log to avoid synchronous file system operations for creating and removing files. Although this improves the speed at which articles can be accepted, it does not greatly reduce the storage requirements.

NewsCache [31] reduces resource consumption by caching news articles. It is designed to replace traditional Usenet servers that are leaf nodes in the news feed graph, allowing them to only retrieve and store articles that are requested by readers. In addition to filling the cache on demand, it can also pre-fetch articles for certain newsgroups. These features are also available as a mode of DNews [54], a commercial high-performance server, which adds the ability to dynamically determine the groups to pre-fetch. Thus, both NewsCache and DNews reduce local bandwidth requirements to be proportional to readership as well as more optimally using local storage. However, they only do this for servers that are leaf nodes and does not help reduce the requirements on upstream servers. By comparison, UsenetDHT also reduces resource requirements for those servers that are not leaf nodes. Also, standard caches will continue to work with UsenetDHT servers.

The implementation of expiration in DHash draws on experiences from Usenet servers. DHash's storage system is closest to the timehash system used by INN (described in [71]).

Chapter 8

Conclusions

This thesis has presented the design of the Passing Tone maintenance algorithm for consistent hashing systems and shown how to implement it within the DHash distributed hash table. While DHash simplifies the programming interface for developers building large-scale distributed storage systems, Passing Tone enables DHash to efficiently maintain replicas despite transient failures. The resulting system supports a high-performance Usenet server that operates over distributed nodes called UsenetDHT.

We conclude this thesis with some directions for future research and a summary of our main results.

8.1 Future directions

This thesis has focused on providing high-performance storage for replicated, immutable data in order to support a single application, such as UsenetDHT. Thus, this research presents a base for the future development of algorithms and implementations that deal with erasure-coded data, mutable data, and support for multiple applications in distributed hash tables. Separately, Usenet's popularity and performance requirements made it useful as a motivating application: Usenet and the UsenetDHT approach can serve a basis for further research in structuring the implementation of Usenet servers and developing algorithms for distributed applications.

8.1.1 Erasure codes

A common alternative to replication in distributed hash table systems is the use of erasure codes (such as IDA [61]); notable examples of systems employing erasure codes are DHash [17] and Glacier [33]. Erasure codes provide a higher degree of durability for a given amount of storage [87]. By breaking an object into fragments of which only a subset are needed for reads, the same degree of durability can be provided with fewer bits stored. Conversely, with the same amount of bits stored, significantly higher durability and availability can be obtained.

Expressed concretely, the erasure codes have two parameters, l and m, where l is the total number of fragments stored and m is the number required to reconstruct the object. Replication is the special case in which $m = 1$, and l is the number of replicas. The *rate* of coding, $r = \frac{l}{m}$, expresses the amount of redundancy. A replication scheme with three replicas has $m = 1$, $l = 3$, and $r = 3$. Our earlier implementations chose to use a 7-out-of-14 IDA erasure code with $m = 7$, $l = 14$, and $r = 2$ motivated by the goal of minimizing the number of bits needed sent over the network and that needed to get written to disk for a given level of availability [17].

The choice of parameters m and l has three main effects. First, it determines a object's availability when nodes fail [87]. If the probability that any given DHT node is available is p_0, the probability that a object is still available is [7]:

$$p_{avail} = \sum_{i=m}^{l} \binom{l}{i} p_0^i (1-p_0)^{l-i}$$

Second, increasing m is likely to increase the fetch latency, since more nodes must be contacted to complete a read (increasing the probability that one of them will be slow). This interacts with the choice of l which can be used to increase the number of options from which to read. Naturally, there is a trade-off because increasing l (and r) means that more hosts must be contacted before a write can complete. Third, increasing r increases the amount of communication required to write a object to the DHT.

Replication makes maintenance easier—repairs in Passing Tone simply require performing a read from one other node. In a system with erasure coding, it is not possible to generate a new unique fragment without the whole object. Thus, the cost of creating a new fragment is higher than the cost of simply transferring a single fragment. It may be that this cost difference (which is what replica maintenance costs) is not significant but it may be that there are optimizations to be achieved in how repairs are handled. Rodrigues and Liskov [67] discuss the trade-offs of using erasure codes in different environments: they conclude that simple replication is likely to be more effective when operating in relatively stable environments like PlanetLab, but erasure coding provides better durability in high churn environments.

We began to evaluate the performance aspects of erasure coding in simulation in [13] but have yet to verify their impact in a real deployment. Future work could seek to understand the impact of erasure coding on maintenance, and whether it can be successfully deployed for an application such as UsenetDHT.

8.1.2 Mutable data

While immutable data is a powerful basis for constructing applications, many systems do prefer in-place mutation or can benefit from it. In fact, DHash has always had rudimentary support for mutable objects. The current code-base has basic support for objects signed by a public key and those that are completely unverified.

Unlike the content-hashed objects keyed by the hash of the object data itself, KEYHASH objects are keyed by a salted hash of a public signature key. The use of a public key allows

for authenticated updates; CFS [16] and Ivy [53] both take advantage of this to provide a well-known rendezvous point for the root of various data-structures. The salt allows for multiple such data-structures to be created for a single public key.

For applications where this may even be too restrictive, DHash provides objects that are not verified by the system at all: NOAUTH objects. These are free-form objects that do not place any constraints on the key or object data. Applications are free to choose any keys (though they are recommended to at least hash their raw keys to ensure uniform distribution); no data integrity is guaranteed by the system. Applications must provide their own integrity checks.

NOAUTH objects may be updated: however, since these updates can not be verified by the DHT nodes themselves, all updates are simply accumulated into a set of updates. The application must handle interpreting these objects and resolving any conflicts that may arise. This requirement complicates maintenance.

Current state-of-the-art DHT systems that support mutation provide eventual consistency [19, 66]. In this view, each mutable object can be viewed as a set of immutable objects; maintenance ensures that these sets are eventually identical across all nodes in the replica set. Applications then interpret this set as a mutable object. For example, each individual write could be treated as a patch to the previous version, in the style of version control systems like RCS or Mercurial [49].

However, mutability poses a challenge for synchronization: all versions of a mutable object share the same logical identifier. Current synchronization algorithms can not distinguish between different versions of the same object. In order to ensure that all nodes converge to the latest version of an object, it is necessary to distinguish these versions in some way for the synchronization system. DHash solves this problem by stealing the low 32-bits from the identifier space to distinguish writes under the same key. For synchronization purposes, the low 32-bits of the object identifier are replaced with an identifier nominally unique to the version of the object. This can be a CRC of the ordered set of writes. When synchronizing, two nodes with different versions will manifest as each missing some key that the other has. All that remains is to actually exchange objects.

While we have a basic implementation of this in our current code-base, it has not been tested in a real system; for example, we do not know how quickly we can converge or how frequently write conflicts might arise, since UsenetDHT does not require mutable data structures. The OverCite application would be an ideal application for future testing and development [79].

8.1.3 Multiple applications

DHash's storage API is independent of application; thus it is not possible in DHash to isolate multiple applications using a single DHT as OpenDHT does [63]. Even adopting OpenDHT's techniques, having multiple applications on a single DHT raises questions of security and also of incentives, particularly with respect to sharing system resources.

This thesis ignores problems of security by requiring that all parties participating in the DHT (for example, to run UsenetDHT) cooperate. However, multiple applications will

require that a distributed hash table include many nodes who have different interests and goals. Such nodes may be mutually distrusting and some nodes may behave maliciously. In earlier work, we identified a number of the security issues related to routing, such as cases where an attacker can prevent clients from reading specific objects [74]. Castro *et al.*'s secure routing system presents a number of techniques for avoiding these problems, but largely depends on node identifier assignment by a trusted authority [9]. When parties are mutually distrustful, alternate techniques may need to be considered. A fruitful line of research would be to explore techniques such as those recently proposed by Lesniewski-Laas [44] or Haeberlen *et al.* [32] in the context of DHash and workloads such as UsenetDHT.

Another issue commonly facing peer-to-peer systems is that of providing incentives for users to behave fairly relative to other users. In the context of DHTs, applications sharing a single DHash deployment will need to share available storage and network capacity. The current design provides no incentives for applications to use the DHT fairly, relative to other applications. For example, a bibliographic database such as OverCite [79] might set an unlimited expiration time for all records, steadily consuming resources and limiting the storage capacity available for UsenetDHT. On the other hand, UsenetDHT's constant read and write workload may result in little network (or disk) bandwidth available for servicing OverCite requests. It may be that certain applications are incompatible with shared infrastructure, or that shared infrastructure (e.g., Amazon's S3) must be provisioned and priced properly so that it can meet the requirements of multiple applications.

8.1.4 Usenet and UsenetDHT

As UsenetDHT deployments grow, sites may wish to partition storage responsibility by newsgroup. The current design allows sites to partition newsgroups only at the level of metadata. Karger and Ruhl have proposed a mechanism for partitioning Chord into heterogeneous subgroups [39]. This technique, applied to the underlying Chord ring in DHash, could allow specific classes of newsgroups (e.g., binary newsgroups versus text newsgroups) to be assigned to specific sites, or specific hosts at a given site. As yet, to our knowledge, the subgroup formation protocol has not been implemented or evaluated in practice. To deploy the Karger-Ruhl protocol for storage would also require that adaptations be developed to make Passing Tone aware of the different subgroups and respect them during maintenance.

A recent proposal calls for a return to distributed file systems as a building block for distributed systems and an end to the cycle of distributed applications continually "re-inventing the wheel" for storage [80]. The authors propose the use of *filesystem cues* to allow applications to specify semantics for failure handling and other problems of distributed applications. Successful re-use of an existing Usenet server on top of this system, called WheelFS, would lend credence to the utility of a common storage approach with a file system interface.

8.2 Summary

This thesis showed how to perform efficient maintenance of immutable data to provide durability in a distributed hash table. The Passing Tone algorithm demonstrates how to overcome challenges presented by transient and permanent failures, limited disk capacity, and how to arrange nodes to minimize coordination and improve repair performance.

Transient failures are dealt with by maintaining and re-integrating extra replicas that act as a buffer between failures and additional repairs. Passing Tone uses successor placement to re-integrate these extra replicas without having to maintain explicit state about their locations. Using predecessor lists, each Passing Tone node knows the precise range of keys that it is responsible for at any time. Using this information, the local and global maintenance components of Passing Tone ensure that the objects are stored in their replica sets with at least r_L copies. The global maintenance component copies objects that have perhaps lost their place; the local maintenance component ensures that all current nodes in the replica set have the objects they should.

Both local and global maintenance require only pairwise synchronizations; no other nodes need be consulted in order to make maintenance decisions. This keeps the resources needed to perform maintenance low and enables decisions to be made quickly. Because each node is responsible for copying objects to itself, it is not dependent on other nodes to ensure that it has the objects it should.

Passing Tone uses expiration times to limit the amount of time that it must spend maintaining the durability of objects. The expiration time is set by the application based on its durability requirements. Our implementation of Passing Tone in DHash efficiently stores objects based on expiration times. Proper arrangement of objects on disk make it fast to both add and remove objects.

This implementation is able to directly support UsenetDHT, a DHT-based Usenet server, and process 2.5 Mbyte/s in new articles; in the steady state, almost no extra bandwidth is used for identifying objects to be repaired. In the case of failure, Passing Tone responds quickly to failures, transferring data across multiple nodes. Additionally, experiments show that DHash scales as additional resources are added.

The source code for Passing Tone and DHash is licensed under the MIT X11 license and is available online at `http://pdos.csail.mit.edu/chord/`.

Bibliography

[1] ANDERSEN, D. *Improving End-to-End Availability Using Overlay Networks*. PhD thesis, Massachusetts Institute of Technology, 2004. (Referenced on page 14.)

[2] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. Resilient overlay networks. In *Proc. of the 18th ACM Symposium on Operating System Principles* (Oct. 2001). (Referenced on pages 16, 35, and 36.)

[3] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. Serverless network file systems. *ACM Transactions on Computer Systems 14*, 1 (1996), 41–79. (Referenced on page 73.)

[4] BARBER, S. Common NNTP extensions. RFC 2980, Network Working Group, Oct. 2000. (Referenced on pages 12 and 55.)

[5] BHAGWAN, R., SAVAGE, S., AND VOELKER, G. Understanding availability. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems* (Feb. 2003). (Referenced on pages 22 and 60.)

[6] BHAGWAN, R., TATI, K., CHENG, Y.-C., SAVAGE, S., AND VOELKER, G. M. Total Recall: System support for automated availability management. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004). (Referenced on pages 14, 15, 26, 29, and 37.)

[7] BLAKE, C., AND RODRIGUES, R. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of the 9th Workshop on Hot Topics in Operating Systems* (May 2003). (Referenced on pages 14, 60, 71, and 78.)

[8] BORMANN, C. The Newscaster experiment: Distributing Usenet news via many-to-more multicast. `http://citeseer.nj.nec.com/251970.html`. (Referenced on page 75.)

[9] CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. S. Secure routing for structured peer-to-peer overlay networks. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation* (Dec. 2002). (Referenced on page 80.)

[10] CASTRO, M., DRUSCHEL, P., HU, Y. C., AND ROWSTRON, A. Exploiting network proximity in peer-to-peer overlay networks. Tech. Rep. MSR-TR-2002-82, Microsoft Research, June 2002. (Referenced on page 72.)

[11] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20*, 4 (2002), 398–461. (Referenced on page 73.)

[12] CATES, J. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003. (Referenced on pages 14, 16, 27, 30, 34, 44, 71, and 74.)

[13] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *Proc. of the 3rd Symposium on Networked Systems Design and Implementation* (May 2006). (Referenced on pages 23, 29, 30, 35, 44, 56, 71, 74, 75, and 78.)

[14] DABEK, F. *A Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, 2005. (Referenced on pages 16, 25, 37, 46, 50, and 71.)

[15] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. Vivaldi: A decentralized network coordinate system. In *Proc. of the 2004 ACM SIGCOMM* (Portland, OR, Aug. 2004). (Referenced on pages 46 and 50.)

[16] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating System Principles* (Oct. 2001). (Referenced on pages 11, 12, 16, 24, 25, 45, 63, and 79.)

[17] DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. Designing a DHT for low latency and high throughput. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004). (Referenced on pages 12, 15, 16, 19, 44, 46, 50, 71, 77, and 78.)

[18] DABEK, F., ZHAO, B., DRUSCHEL, P., KUBIATOWICZ, J., AND STOICA, I. Towards a common API for structured peer-to-peer overlays. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems* (Feb. 2003). (Referenced on page 50.)

[19] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM Symposium on Operating System Principles* (Oct. 2007). (Referenced on pages 11, 25, 26, 27, 48, 72, and 79.)

[20] FORD, B. Structured streams: a new transport abstraction. In *Proc. of the 2007 ACM SIGCOMM* (Aug. 2007). (Referenced on page 47.)

[21] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with Coral. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004). (Referenced on pages 12, 14, and 72.)

[22] FRITCHIE, S. L. The cyclic news filesystem: Getting INN to do more with less. In *Proc. of the 9th LISA* (1997), pp. 99–112. (Referenced on page 75.)

[23] GHANE, S. Diablo statistics for spool-1t2.cs.clubint.net (Top1000 #24). `http://usenet.clubint.net/spool-1t2/stats/`. Accessed 18 March 2008. (Referenced on pages 20 and 61.)

[24] GHANE, S. The official Top1000 Usenet servers page. `http://www.top1000.org/`. Accessed 13 September 2007. (Referenced on pages 61 and 65.)

[25] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. of the 2003 19th ACM Symposium on Operating System Principles* (Bolton Landing, NY, Oct. 2003). (Referenced on pages 11, 26, and 73.)

[26] GIGANEWS. 1 billion usenet articles. `http://www.giganews.com/blog/2007/04/1-billion-usenet-articles.html`, Apr. 2007. (Referenced on page 20.)

[27] GODFREY, P. B., AND STOICA, I. Heterogeneity and load balance in distributed hash tables. In *Proc. of the 24th Conference of the IEEE Communications Society (Infocom)* (Mar. 2005). (Referenced on pages 13 and 25.)

[28] GRADWELL.COM. Diablo statistics for news-peer.gradwell.net. `http://news-peer.gradwell.net/`. Accessed 30 September 2007. (Referenced on page 21.)

[29] GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the System R database manager. *ACM Computing Surveys 13*, 2 (1981), 223–242. (Referenced on page 73.)

[30] GRIBBLE, S., BREWER, E., HELLERSTEIN, J., AND CULLER, D. Scalable, distributed data structures for internet service construction. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation* (Oct. 2000). (Referenced on page 73.)

[31] GSCHWIND, T., AND HAUSWIRTH, M. NewsCache: A high-performance cache implementation for Usenet news. In *Proc. of the 1999 USENIX Annual Technical Conference* (June 1999), pp. 213–224. (Referenced on page 75.)

[32] HAEBERLEN, A., KUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical accountability for distributed systems. In *Proc. of the 21st ACM Symposium on Operating System Principles* (Oct. 2007). (Referenced on page 80.)

[33] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005). (Referenced on pages 16, 27, 29, 34, 38, 72, 73, 74, and 77.)

[34] HAMANO, J. C., AND TORVALDS, L. Git—fast version control system. `http://git.or.cz/`, 2007. (Referenced on page 12.)

[35] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE, JR, J. W. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation* (Oct. 2000). (Referenced on page 60.)

[36] KAASHOEK, F., AND KARGER, D. Koorde: A simple degree-optimal distributed hash table. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems* (Feb. 2003). (Referenced on page 50.)

[37] KANTOR, B., AND LAPSLEY, P. Network news transfer protocol. RFC 977, Network Working Group, Feb. 1986. (Referenced on pages 12 and 55.)

[38] KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654–663. (Referenced on page 24.)

[39] KARGER, D., AND RUHL, M. Diminished Chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems* (Feb. 2004). (Referenced on pages 59 and 80.)

[40] KARGER, D., AND RUHL, M. Simple efficient load-balancing algorithms for peer-to-peer systems. *Theory of Computing Systems 39*, 6 (2006), 787–804. (Referenced on page 25.)

[41] KARP, B., RATNASAMY, S., RHEA, S., AND SHENKER, S. Spurring adoption of DHTs with OpenHash, a public DHT service. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems* (Feb. 2004). (Referenced on page 14.)

[42] KOTLA, R., ALVISI, L., AND DAHLIN, M. SafeStore: A durable and practical storage system. In *Proc. of the 2007 Usenix Annual Technical Conference* (June 2007). (Referenced on page 72.)

[43] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proc. of the 7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems* (1996), pp. 84–92. (Referenced on page 73.)

[44] LESNIEWSKI-LAAS, C. A Sybil-proof one-hop DHT. In *Proc. of the Workshop on Social Network Systems* (Glasgow, Scotland, April 2008). (Referenced on page 80.)

[45] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proc. of the 6th Symposium on Operating Systems Design and Implementation* (Dec. 2004). (Referenced on page 12.)

[46] LI, J., STRIBLING, J., MORRIS, R., AND KAASHOEK, M. F. Bandwidth-efficient management of DHT routing tables. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005). (Referenced on pages 13, 15, 19, 44, and 50.)

[47] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the Harp file system. In *Proc. of the 13th ACM Symposium on Operating System Principles* (Oct. 1991), pp. 226–38. (Referenced on page 73.)

[48] LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. LH*—a scalable, distributed data structure. *ACM Transactions on Database Systems 21*, 4 (1996), 480–525. (Referenced on page 71.)

[49] MACKALL, M. Towards a better SCM: Revlog and Mercurial. In *Proc. of the 2006 Linux Symposium* (Ottawa, Canada, July 2006). (Referenced on page 79.)

[50] MAZIÈRES, D. A toolkit for user-level file systems. In *Proc. of the 2001 Usenix Annual Technical Conference* (June 2001). (Referenced on page 43.)

[51] MINSKY, Y., TRACHTENBERG, A., AND ZIPPEL, R. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory 49*, 9 (Sept. 2003), 2213–2218. Originally published as Cornell Technical Report 2000-1796. (Referenced on pages 16, 34, and 74.)

[52] MISLOVE, A., POST, A., HAEBERLEN, A., AND DRUSCHEL, P. Experiences in building and operating a reliable peer-to-peer application. In *Proc. of the 1st EuroSys Conference* (April 2006). (Referenced on page 34.)

[53] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A read/write peer-to-peer file system. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation* (Dec. 2002). (Referenced on pages 11, 12, and 79.)

[54] NETWIN. DNews: Unix/Windows Usenet news server software. `http://netwinsite.com/dnews.htm`. Accessed 9 November 2003. (Referenced on page 75.)

[55] NIXON, J., AND D'ITRI, M. Cleanfeed: Spam filter for Usenet news servers. `http://www.exit109.com/~jeremy/news/cleanfeed/`. Accessed on 15 February 2004. (Referenced on page 54.)

[56] PARK, K. S., AND PAI, V. CoMon: a mostly-scalable monitoring system for PlanetLab. *ACM SIGOPS Operating Systems Review 40*, 1 (Jan. 2006), 65–74. `http://comon.cs.princeton.edu/`. (Referenced on pages 35 and 66.)

[57] PATTERSON, D., GIBSON, G., AND KATZ, R. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the ACM SIGMOD International Conference on Management of Data* (June 1988). (Referenced on page 73.)

[58] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the 1st Workshop on Hot Topics in Networking* (Oct. 2002). (Referenced on page 36.)

[59] PETERSON, L., BAVIER, A., FIUCZYNSKI, M. E., AND MUIR, S. Experiences building PlanetLab. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, Nov. 2006). (Referenced on pages 21 and 35.)

[60] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival data storage. In *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST)* (Jan. 2002). (Referenced on page 12.)

[61] RABIN, M. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36*, 2 (Apr. 1989), 335–348. (Referenced on pages 44 and 77.)

[62] RAMASUBRAMANIAN, V., AND SIRER, E. G. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004). (Referenced on page 72.)

[63] RHEA, S. *OpenDHT: A Public DHT Service*. PhD thesis, University of California, Berkeley, 2005. (Referenced on pages 35, 48, 71, and 79.)

[64] RHEA, S., CHUN, B.-G., KUBIATOWICZ, J., AND SHENKER, S. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *Proc. of the 2nd Workshop on Real Large Distributed Systems* (Dec. 2005). (Referenced on pages 34 and 46.)

[65] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)* (Apr. 2003). (Referenced on pages 11 and 71.)

[66] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. OpenDHT: A public DHT service and its uses. In *Proc. of the 2005 ACM SIGCOMM* (Aug. 2005). (Referenced on pages 12, 71, and 79.)

[67] RODRIGUES, R., AND LISKOV, B. High availability in DHTs: Erasure coding vs. replication. In *Proc. of the 4th International Workshop on Peer-to-Peer Systems* (Feb. 2005). (Referenced on page 78.)

[68] ROLFE, A. Private communication, 2007. (Referenced on pages 13 and 61.)

[69] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symposium on Operating System Principles* (Oct. 2001). (Referenced on pages 11 and 71.)

[70] SAITO, Y., FRŒLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. FAB: building distributed enterprise disk arrays from commodity components. In *Proc. of the 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, 2004), ACM Press, pp. 48–58. (Referenced on page 73.)

[71] SAITO, Y., MOGUL, J. C., AND VERGHESE, B. A Usenet performance study. `http://www.research.digital.com/wrl/projects/newsbench/usenet.ps`, Nov. 1998. (Referenced on pages 45, 61, and 75.)

[72] SIT, E., DABEK, F., AND ROBERTSON, J. UsenetDHT: A low overhead Usenet server. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems* (Feb. 2004). (Referenced on pages 12 and 16.)

[73] SIT, E., HAEBERLEN, A., DABEK, F., CHUN, B.-G., WEATHERSPOON, H., KAASHOEK, F., MORRIS, R., AND KUBIATOWICZ, J. Proactive replication for data durability. In *Proc. of the 5th International Workshop on Peer-to-Peer Systems* (Feb. 2006). (Referenced on page 74.)

[74] SIT, E., AND MORRIS, R. Security considerations for peer-to-peer distributed hash tables. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002). (Referenced on page 80.)

[75] SIT, E., MORRIS, R., AND KAASHOEK, M. F. UsenetDHT: A low overhead design for Usenet. In *Proc. of the 5th Symposium on Networked Systems Design and Implementation* (Apr. 2008). (Referenced on page 12.)

[76] SONG, Y. J., RAMASUBRAMANIAN, V., AND SIRER, E. G. Optimal resource utilization in content distribution networks. Computing and Information Science Technical Report TR2005-2004, Cornell University, Nov. 2005. (Referenced on page 72.)

[77] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the 2001 ACM SIGCOMM* (Aug. 2001). An extended version appears in ACM/IEEE Trans. on Networking. (Referenced on pages 19 and 24.)

[78] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking* (2002), 149–160. (Referenced on pages 13, 44, 50, and 56.)

[79] STRIBLING, J., LI, J., COUNCILL, I. G., KAASHOEK, M. F., AND MORRIS, R. Exploring the design of multi-site web services using the OverCite digital library. In *Proc. of the 3rd Symposium on Networked Systems Design and Implementation* (May 2006). (Referenced on pages 79 and 80.)

[80] STRIBLING, J., SIT, E., KAASHOEK, M. F., LI, J., AND MORRIS, R. Don't give up on distributed file systems. In *Proc. of the 6th International Workshop on Peer-to-Peer Systems* (Feb. 2007). (Referenced on page 80.)

[81] SWARTZ, K. L. Forecasting disk resource requirements for a usenet server. In *Proc. of the 7th USENIX Large Installation System Administration Conference* (1993). (Referenced on page 20.)

[82] TOLIA, N., KAMINSKY, M., ANDERSEN, D. G., AND PATIL, S. An architecture for Internet data transfer. In *Proc. of the 3rd Symposium on Networked Systems Design and Implementation* (May 2006). (Referenced on page 47.)

[83] Network status page. `http://www.usenetserver.com/en/networkstatus.php`. Accessed 30 September 2007. (Referenced on pages 12 and 21.)

[84] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation* (Dec. 2004). (Referenced on page 73.)

[85] VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. TCP Nice: A mechanism for background transfers. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation* (Dec. 2002). (Referenced on page 47.)

[86] WEATHERSPOON, H., EATON, P., CHUN, B.-G., AND KUBIATOWICZ, J. Antiquity: Exploiting a secure log for wide-area distributed storage. In *Proc. of the 2nd ACM European Conference on Computer Systems (Eurosys 2007)* (Mar. 2007). (Referenced on pages 12, 29, and 72.)

[87] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002). (Referenced on pages 77 and 78.)

[88] WOLLMAN, G. Private communication, 2007. (Referenced on pages 13 and 61.)