Robust and Efficient Data Management for a Distributed Hash Table

by

Josh Cates

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

©2003 Josh Cates. All rights reserved.

The author hereby grants M.I.T. permission to reproduce and distributed publicly
paper and electronic copies of this thesis and to grant others the right to do so.

Author . . . . . .
Department of Electrical Engineering and Computer Science
16 May, 2003

Certified by . . . . . .
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . . . .
Robert Morris
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

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

Robust and Efficient Data Management for a Distributed Hash Table

by

Josh Cates

Submitted to the Department of Electrical Engineering and Computer Science
on 16 May, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis presents a new design and implementation of the DHash distributed hash table based on erasure encoding. This design is both more robust and more efficient than the previous replication-based implementation [15].

DHash uses erasure coding to store each block as a set of fragments. Erasure coding increases availability while saving storage and communication costs compared to a replication based design. DHash combines Chord's synthetic coordinates with the the set of fragments to implement server selection on block retrieval.

DHash enhances robustness by implementing efficient fragment maintenance protocols. These protocols restore missing or misplaced fragments caused by hosts joining and leaving the system.

Experiments with a 270-node DHash system running on the PlanetLab [1] and RON [4] testbeds show that the changes to DHash increase the rate at which the system can fetch data by a factor of six, and decrease the latency of a single fetch by more than a factor of two. The maintenance protocols ensure that DHash is robust without penalizing performance. Even up to large database size, the per host memory footprint is less than 10 MB and the per host network bandwidth is under 2 KB/sec over a wide range of system half-lives.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Robert Morris
Title: Assistant Professor of Computer Science and Engineering

"Of the gladdest moments in human life, methinks, is the departure upon a distant journey into unknown lands. Shaking off with one mighty effort, the fetters of Habit, the leaden weight of Routine, the cloak of many cares and the slavery of Home, one feels once more happy."

Sir Richard Burton - Journal Entry - 1856

Acknowledgments

I would like to thank Prof. Frans Kaashoek for his enduring support and guidance. From when I was an undergraduate UROPer until today, he has been a constant source of encouragement. His efforts have afforded me the opportunity to work with an unforgettable group of faculty and students. In particular, this thesis benefits greatly from his insights, suggestions, and patient revisions.

This thesis is based on joint work with the members of the Chord project. The primary contribution of this thesis is the use of erasure coding and fragment repair within DHash, which couldn't have been designed and implemented without all of work of Russ Cox, Frank Dabek, Prof. Frans Kaashoek, Prof. Robert Morris, James Robertson, Emil Sit and Jacob Strauss. Discussions with them have improved the system considerably. Prof. Morris deserves to be thanked for his especially lucid design feedback. Emil Sit coded significant parts of the work described in this thesis.

A huge thanks is due to all the members of the PDOS research group for making work a lot of fun. They have made for a collection of great memories. I'm very grateful to Chris Zimman who opened my eyes to MIT in the first place. Finally, I'd like to thank my parents for their constant encouragement.

This research was partially supported by the IRIS project (`http://project-iris.net/`), by the National Science Foundation under Cooperative Agreement No. ANI-0225660, the Oxygen project (`http://oxygen.lcs.mit.edu/`), by the RON project (`http://ron.lcs.mit.edu/`), and by the Defense Advanced Research Projects Agency (DARPA).

Contents

1 Introduction **11**

1.1 Peer-to-peer Off-site Backup 13

1.2 Background: Chord 15

1.2.1 Chord API 16

1.2.2 Synthetic Coordinates 17

1.3 Thesis Overview 18

2 DHash: a distributed hash table **19**

2.1 DHash API 20

2.2 Block Availability 20

2.3 Block Insert: `put(`k`, `b`)` 22

2.4 Block Fetch: `get(`k`)` 22

3 Fragment Maintenance **25**

3.1 Global DHash Maintenance 26

3.2 Local DHash Maintenance 29

4 Database Synchronization **31**

4.1 Approach 31

4.2 Database Properties 33

4.3 Database Index Format 33

4.3.1 Index Insertion 35

4.4 Network Protocol 36

5 Evaluation **41**

5.1 Basic Performance 41

5.2 Fetch Latency 42

5.3 Synchronization Dynamics 44

5.4 Ideal State Maintenance 45

5.5 Effect of Half-Life 47

5.6 Synchronization Overhead 47

5.7 Memory Usage 49

5.8 Fault Tolerance 50

6 Related Work **53**

6.1 Cooperative Backup 54

6.2 Server Selection and Spreading 54

6.3 Coding 55

6.4 Replica Synchronization 55

7 Summary **57**

7.1 Conclusions 57

7.2 Future Work 57

Chapter 1

Introduction

DHTs have been proposed as a way to simplify the construction of large-scale distributed applications (e.g., [2, 26]). DHTs[1] store blocks of data on a collection of nodes spread throughout the Internet. Each block is identified by a unique key. The goals of these DHTs are to spread the load of storing and serving data across all the nodes and to keep the data available as nodes join and leave the system.

This thesis presents a new design, based on erasure coding, for distributing and storing blocks within DHash, an existing DHT implemention. These changes make DHash a robust, efficient and practical DHT for demanding applications such as cooperative backup [13, 18, 24]. Such an application requires that the DHT keep data available despite faults and that the DHT efficiently serve bulk data (unlike, for example, a naming system).

Like many fault-tolerant systems, DHash uses erasure coding to increase availability with relatively little cost in extra storage and communication. DHash stores each block as a set of erasure encoded fragments; the block itself is never stored by the system. This work

[1]The DHT community has some disagreement about what the best API for a DHT is [16, 19] and the OceanStore project uses a DOLR instead of a DHT [33], but for most of the contributions of this thesis these distinctions are not very significant.

extends the previous DHash system, which replicated blocks for fault-tolerance [15].

The main contribution of this thesis is the way Dhash combines erasure encoded storage with the other techniques and properties of Chord to provide robust and efficient operation. These other techniques include proximity routing, server selection, and successor lists. As we demonstrate through experiments with DHash implementation, the synergy between the techniques makes them more effective as a collection than individually.

The second contribution of this thesis is the design and implementation of a pair of fragment maintenance protocols that ensure DHash is robust: i.e., that each inserted block can subsequently be retrieved, even if nodes join and leave the system. These protocols restore any fragments which get destroyed or misplaced when hosts join or leave the system. The challenge is to make these protocols as efficient as possible given a very general failure model.

The final contribution of this thesis is a detailed evaluation of DHash. Our evaluation of the previous, replication based DHash running on the PlanetLab [1] and RON [4] test-beds, makes clear that it is inadequate to support data-intensive applications which require high data availability. DHash, with our changes, offers increased block-download throughput, reduced block fetch latency, and improved availability for a given space replication budget.

1.1 Peer-to-peer Off-site Backup

In order to help guide design decisions for DHash, we implemented a cooperative off-site backup system. The off-site backups are intended to complement conventional tape or disk-to-disk backups by adding an extra level of availability and providing a browseable archive of backups. The off-site backup system can be used alone if desired.

The off-site backup system's goals are to support recovery after a disaster by keeping snapshots of file systems at other Internet sites. The system spreads the data over many sites in order to balance storage and network load. This striping also allows very large file systems to be backed up onto a set of hosts with individually limited disk space. The backup system performs daily incremental backups; each new backup shares storage with the unchanged part of the previous backups.

The intended users of the backup system are informal groups of people at geographically distributed sites who know each other; for example, colleagues at different university computer science departments. Each site is expected to make available spare disk space on workstations. These workstations are likely to be reasonably reliable and have fairly fast network connections.

Since the backup system sends file system copies over the Internet, communication performance is important; it must be possible to back up a full day's incremental changes to a typical server file system in a few hours. Performance is more sensitive to network throughput than to latency, since the backup system usually has large quantities of data that can be sent concurrently. Storing and fetching data are both important because the backup system allows users to browse the archive of old snapshots.

At a low level, the backup system works in units of 8192-byte disk blocks; it scans the disk to be backed up, looking for blocks that have changed since the last backup and

ignoring blocks that are not allocated. It inserts each new block into DHash, using the hash of the block's contents as its key. The system builds a tree with the blocks as leaves in order to be able to map disk block numbers to block contents; the tree's interior nodes are also stored in DHash.

Each daily backup has its own tree, though in practice each tree shares most of its DHash storage with previous backups. The block orientation of this scheme is a good match for the block interface of DHash. Because the backup system performs sharing at the block level, it is not necessary to search for a machine whose file system is similar overall; while such a file system may exist when a workstation is being backed up [13], the backup of a large server holding home directories is unlikely to find such a partner.

In summary, the backup system places the following requirements on DHash: 1) high availability, 2) high throughput for bulk transfers, 3) low latency for interactive browsing of the archived backups, and 4) good support for block-size operations.

1.2 Background: Chord

DHash uses Chord [38] to help determine on which host to store each piece of data. Chord implements a hash-like lookup operation that maps 160-bit data keys to hosts. Chord assigns each host an identifier drawn from the same 160-bit space as the keys. This identifier space can be viewed as a circle, in which the highest identifier is followed by zero. Chord maps each key to the host whose identifier most closely follows the key.

Each Chord host maintains information about a number of other hosts, to allow it to efficiently map keys to hosts and to allow it to tolerate failures. Chord ensures that each host knows the identity (IP address, Chord identifier, and synthetic coordinates) of its *successor*: the host with the next highest identifier. This knowledge organizes the hosts into a circular linked list sorted by identifier.

In order to maintain the integrity of this organization if nodes fail, each node actually maintains a *successor list*, which contains the identities to the r hosts that immediately follow the host in the identifier circle. If a node's successor is not responsive, the node replaces it with the next entry in its successor list. The Chord implementation used in this paper uses successor lists of length $r = 16$; this is $2 \log_2 N$ for the system evaluated in Section 5, as recommended in the Chord design [38].

The lookup to map a key to a host could in principle be implemented in linear time by following the successor pointers. Chord builds a routing table, called the *finger table*, that allows it to perform lookups in $O(\log N)$ time, where N is the number of hosts in the Chord system. The finger table for a node n contains $\log N$ entries that point to hosts at power-of-two distances ahead of n along the identifier circle.

A Chord host periodically checks the validity of its finger table and successor list entries in a process called *stabilization*. This process allows Chord to adapt to failed hosts and to

newly joining hosts. Chord also periodically tries to contact hosts that were alive in the past, but are no longer reachable; this allows Chord to notice when a network partition has healed.

1.2.1 Chord API

Table 2.1 shows the external API Chord exposes to DHash.

Function	**Description**
`get_successor_list(`n`)`	Contacts Chord node n and returns n's successor list. Each node in the list includes its Chord ID, IP address and synthetic coordinates.
`lookup(`k`, `m`)`	Returns a list of at least m successors of key k. Each node in the list includes its Chord ID, IP address and synthetic coordinates.

Table 1.1: Chord API

`get_successor_list(`n`)` is a simple accessor method for the Chord node n. It is implemented as a single network RPC call.

`lookup(`k`, `m`)`, on the other hand, must send $O(\log N)$ RPCs in order to determine the m successors of key k. The value of m affects the latency of the lookup. Higher values of m constrain the lookup routing to travel through specific – potentially high latency – nodes. For example, when $m = 16$, the lookup finds the exact predecessor of k and request its successor list.

Smaller values of m permit flexibility in the routing which allows high latency nodes to be avoided. For example, the two Chord nodes preceding key k both have successor lists which contain at least 15 successors of k. So for $m = 15$, the lookup routing can choose the predecessor with lower estimated latency. A node uses synthetic coordinates to estimate the latency to another node.

1.2.2 Synthetic Coordinates

Synthetic coordinates allow Chord to predict the latency between nodes. The predicted latency in microseconds is equal to the Euclidean distance between the nodes' coordinates. The main advantage of synthetic coordinates is that nodes can predict the latency to nodes with which it has never communicated directly: node X need only know node Y's coordinates to estimate the latency.

When a Chord node first joins the system it chooses random coordinates for itself. Each Chord node continually improves its own coordinates by participating in a distributed machine learning algorithm, called Vivaldi, based on [14]. Each time a Chord node makes an RPC request to another node, it measures the network latency to the node. All RPC responses include the responding node's current coordinates. The requesting node refines its coordinates based on the latency measurement and the responding node's coordinates. Synthetic coordinates do not generate any probe traffic themselves, but merely piggy back on existing Chord stabilization traffic.

Both Chord and DHash use synthetic coordinates to reduce latency by performing server selection. When performing a `lookup()`, Chord uses coordinates to avoid routing through high latency nodes. DHash preferentially fetches data from low latency nodes when there are many possible nodes holding the data (see Section 2.4).

1.3 Thesis Overview

This thesis starts by presenting the redesigned DHash using erasure encoding in Chapter 2. Chapter 3 describes the maintenance of the erasure encoded fragments and Chapter 4 details the database synchronization algorithms used by the maintenance protocols. In Chapter 5, an implementation of the system is evaluated for robustness and performance. Chapter 6 reviews the related work. And finally, Chapter 7 concludes and suggests possible future work for DHash.

Chapter 2

DHash: a distributed hash table

The DHash servers form a distributed hash table, storing opaque blocks of data named by the `SHA-1` hash of their contents. Clients can insert and retrieve blocks from this hash table. The storage required scales as the number of unique blocks, since identical blocks hash to the same server, where they are coalesced.

The hash table is implemented as a collection of symmetric nodes (*i.e.*, each node is no more special in function than any other node). Clients inserting blocks into DHash need not share any administrative relationship with servers storing blocks. DHash servers could be ordinary Internet hosts whose owners volunteer spare storage and network resources. DHash allows nodes to enter or leave the system at any time and divides the burden of storing and serving blocks among the servers.

To increase data availability, DHash splits each block into 14 fragments using the IDA erasure code. Any 7 of these fragments are sufficient to reconstruct the block. DHash stores a block's fragments on the 14 Chord nodes immediately following the block's key. To maintain this proper placement of fragments, DHash transfers fragments between nodes as nodes enter and leave the system.

2.1 DHash API

Table 2.1 shows that the external API exposed by DHash is minimal. There are calls to insert and retrieve a block.

Function	Description
`put(`k`, `b`)`	Stores the block b under the key k, where k = `SHA-1`(b).
`get(`k`)`	Fetches and returns the block associated with the key k.

Table 2.1: DHash API

2.2 Block Availability

Like many fault-tolerant storage systems, DHash uses erasure coding to increase availability with relatively little cost in extra storage and communication. DHash uses the IDA erasure code [31]. Given an original block of size s, IDA splits the block into f fragments of size s/k. Any k *distinct* fragments are sufficient to reconstruct the original block. Fragments are distinct if, in an information theoretic sense, they contain unique information.

IDA has the ability to randomly generate new, probabilistically distinct fragments from the block alone; it does not need to know which fragments already exist. From f randomly generated fragments, any k are distinct with probability greater than $\frac{p-1}{p}$, where p is the characteristic prime of the IDA implementation.

This ability to easily generate new fragments contrasts with Reed-Solomon codes, which generate only a small set of fragments for a given rate. Other codes, such as Tornado codes and On-line codes, are targeted to provide efficiency asymptotically and do not perform well with small blocks.

DHash leverages IDA's ability to generate probabilistically distinct random fragments to easily and efficiently reconstruct a missing fragment (for example, after a machine crash).

Instead of needing to find all existing fragments to ensure that the new fragment is distinct, DHash must only find enough fragments to reconstruct the block, and can then generate a new random fragment to replace the missing fragment.

DHash implements IDA with $f = 14$, $k = 7$ and $p = 65537$. DHash stores a block's fragments on the $f = 14$ immediate successors of the block's key. When a block is originally inserted, the DHash code on the inserting client creates the f fragments and sends them to the first 14 successors (Section 2.3). When a client fetches a block, the client contacts enough successors to find $k = 7$ distinct fragments (Section 2.4). These fragments have a 65536-in-65537 chance of being able to reconstruct the original block. If reconstruction fails, DHash keeps trying with different sets of 7 fragments.

A node may find that it holds a fragment for a block even though it is beyond the 14th successor. If it is the 15th or 16th successor, the node holds onto the fragment in case failures cause it to become one of the 14. Otherwise the node tries to send the fragment to one of the successors (Section 3).

The choice of f and k are selected to optimize for 8192-byte blocks in our system which has a successor list length of $r = 16$. A setting of $k = 7$ creates 1170-byte fragments, which fit inside a single IP packet when combined with RPC overhead. Similarly $f = 14$ interacts well with $r = 16$ by giving the lookups needed for store and fetch the flexibility to terminate at low latency node close to the key, not necessarily exactly at the key's predecessor.

This choice of IDA parameters also gives reasonable fault tolerance: in a system with a large number of nodes and independent failures, the probability that seven or more of a block's fragments will survive after 10% of the nodes fail is 0.99998 [40]. If two complete copies of each block were stored instead, using the same amount of space, the probability would be only 0.99.

2.3 Block Insert: put(k, b)

When an application wishes to insert a new block, it calls the DHash put(k, b) procedure. The DHash code running on the application's node implements put as follows:

```
void
put (k, b)
// place one fragment on each successor
{
   frags = IDAencode (b)
   succs = lookup (k, 14)
   for i (0..13)
     send (succs[i].ipaddr, k, frags[i])
}
```

Figure 2-1: An implementation of DHash's put(k, b) procedure.

The latency of the complete put() operation is likely to be dominated by the maximum round trip time to any of the 14 successors. The Chord lookup is likely to be relatively low latency: proximity routing allows it to contact nearby nodes, and the lookup can stop as soon as it gets to any of the three nodes preceding key k, since the 16-entry successor list of any of those nodes will contain the desired 14 successors of k. The cost of the Chord lookup is likely to be dominated by the latency to the nearest of the three predecessors.

2.4 Block Fetch: get(k)

In order to fetch a block, a client must locate and retrieve enough IDA fragments to reassemble the original block. The interesting details are in how to avoid communicating with high-latency nodes and how to proceed when some fragments are not available.

When a client application calls get(k), its local DHash first initiates a Chord call to lookup(k, 7), in order to find the list of nodes likely to hold the block's fragments. The

lookup call will result in a list of between 7 and 16 of the nodes immediately succeeding key k.

`get()` then chooses the seven of these successors with the lowest latency, estimated from their synthetic coordinates. It sends each of them an RPC to request a fragment of key k, in parallel. For each RPC that times out or returns an error reply, `get()` sends a fragment request RPC to an as-yet-uncontacted successor from the list returned by `lookup()`. If the original call to `lookup()` returned fewer than 7 successors with distinct fragments, `get()` asks one of the successors it knows about for the complete list if it needs to. `get()` asks more successors for fragments if IDA fails to reconstruct the block because the fragments found were not distinct. If it cannot reconstruct the block after talking to the first 14 successors, `get()` returns failure to the application.

Before returning a reconstructed block to the application, `get()` checks that the SHA-1 hash of the block's data is equal to the block's key. If it is not, `get()` returns an error.

An application may occasionally need to repeatedly invoke `get(`k`)` to successfully fetch a given key. When nodes join or leave the system, fragments need to be transferred to the correct successor nodes. If the join or leave rate is high enough fragments may become misplaced and cause a block fetch to fail. This transient situation is repaired by the DHash maintenance algorithm presented in the next section and can be masked by retrying the `get(`k`)` on failure. By retrying, a client will see the semantics that DHash never loses a block and that all blocks are always available except those that have expired.

A persistent retry strategy reflects the assumption that a key that is retrieved is actually stored in DHash. The client using DHash can easily ensure this by recording keys in meta data and only retrieving keys recorded in this meta data.

```
block get (k)
{
   // Collect fragments from the successors.
   frags = []; // empty array
   succs = lookup (k, 7)
   sort_by_latency (succs)

   for (i = 0; i < #succs && i < 14; i++) {
      // download fragment
      <ret, data> = download (key, succ[i])
      if (ret == OK)
        frags.push (data)

      // decode fragments to recover block
      <ret, block> = IDAdecode (frags)
      if (ret == OK)
         return (SHA-1(block) != k) ? FAILURE : block

      if (i == #succs - 1) {
         newsuccs = get_successor_list (succs[i])
         sort_by_latency (newsuccs)
         succs.append (newsuccs)
      }
   }

   return FAILURE
}
```

Figure 2-2: An implementation of the DHash's `get(`k`)` procedure.

Chapter 3

Fragment Maintenance

A DHash system is in the *ideal state* when three conditions hold for each inserted block:

1. **multiplicity:** 14, 15, or 16 fragments exist.
2. **distinctness:** All fragments are distinct with high probability.
3. **location:** Each of the 14 nodes succeeding the block's key store a fragment; the following two nodes optionally store a fragment; and no other nodes store fragments.

The ideal state is attractive since it ensures all block fetches succeed, with high probability. Block inserts preserve the ideal state, since `put(`k`, `b`)` stores 14 distinct fragments of block b at the 14 Chord nodes succeeding key k.

Chord membership changes, such as node joins and node failures, perturb DHash from the ideal state, and can cause block fetches to fail. The location condition is violated when a new node storing no fragments joins within the set of 14 successors nodes of a block's key, since it does not store a fragment of the block. The multiplicity condition can be violated when nodes fail since fragments are lost from the system. The distinctness condition is not affected by node joins or failures.

To restore DHash to the ideal state, DHash runs two maintenance protocols: a local and a global protocol. The local maintenance protocol restores the multiplicity condition by recreating missing fragments. The global maintenance protocol moves misplaced fragments (those that violate the location condition) to the correct nodes, restoring the location condition. The global maintenance protocol also restores the multiplicity conditions. It detects and deletes extra fragments when more than 16 fragments exist for a block.

Since membership changes happen continuously, DHash is rarely or never in the ideal state, but always tends toward it by continually running these maintenance protocols.

The maintenance protocols can restore DHash to its ideal state if there are at least 7 distinct fragments for each block located anywhere in the system, barring any more membership changes. First, the global maintenance protocol will move the misplaced fragments back to their successors, then the local maintenance protocol will recreate missing fragments until there are 14 fragments. If there are fewer than 7 distinct fragments for a block, that block is lost irrevocably.

3.1 Global DHash Maintenance

The global maintenance protocol pushes misplaced fragments to the correct nodes. Each DHash node scans its database of fragments and pushes any fragment that it stores, but which fail the location condition, to one of the fragment's 14 successor hosts. For efficiency, the algorithm processes contiguous ranges of keys at once.

Each DHash host continuously iterates through its fragment database in sorted order. It performs a Chord `lookup()` to learn the 16 successor nodes of a fragment's key. These nodes are the only hosts which should be storing the fragment. If the DHash host is one of these nodes, the host continues on to the next key in the database, as it should be storing

```
global_maintenance (void)
{
   a = myID
   while (1) {
      <key, frag> = database.next(a)
      succs = lookup(key, 16)
      if (myID isbetween succ[0] and succ[15])
         // we should be storing key
         a = myID
      else {
         // key is misplaced
         for each s in succs[0..13] {
            response = send_db_keys (s, database[key .. succs[0]])
            for each key in response.desired_keys
                if (database.contains (key))
                   upload (s, database.lookup (key))
                   database.delete (key)
         }
         database.delete_range ([pred .. succs[0]])
         a = succs[0]
      }
   }
}
```

Figure 3-1: An implementation of the global maintenance protocol.

the key. Otherwise, the DHash host is storing a misplaced fragment and needs to push it to one of the fragment's 14 successors, in order to restore the location condition.

The fragment's 14 successors should also store all keys ranging from the fragment's key up to the Chord ID of the key's immediate successor. Consequently, the DHash host processes this entire key range at once by sending all the database keys in this range to each of the 14 successors. A successor responds with a message that it desires some key, if it is missing the key. In which case, the DHash host sends that fragment to that successor. It also deletes the fragment from its database to ensure that the fragment is only sent to exactly one other host, with high probability; otherwise the distinctness condition would be violated.

After offering all the keys to all the successors, the DHash node deletes all remaining keys in the specified range from the database. These keys are safe to delete because they have been offered to all the successors, but were already present on each successor. The DHash node continues sweeping the database from the end of the key range just processed.

The call to `database.next(a)` skips the ranges of the key space for which no blocks are in the database. This causes the sweep (i.e., the number of iterations of the `while` loop needed to process the entire key space) to scale as the number of misplaced blocks in the database, not as the number of nodes in the entire Chord ring.

The global maintenance protocol adopts a push-based strategy where nodes push misplaced fragments to the correct locations. This strategy contrasts with pull-based strategies [15] where each node contacts other nodes and pulls fragments which it should store. In general, we've found that push-based strategies are more robust under highly dynamic membership change rates, where pull-based strategies would prove fragile or inefficient.

To be able to recover from an arbitrary set of host failures, a pull-based strategy must pull from all nodes in the entire system. For example, consider a network partition that splits the Chord ring into a majority ring and a very small minority ring. All the block fragments that were inserted into the minority ring during the network partition are almost certainly misplaced when the rings merge. With a pull-based strategy, the correct nodes in the merged ring will have to pull blocks from the nodes from the minority ring that just merged. In essence, each node needs to know when each other node joins the ring, or else it must sweep the entire Chord ring trying to pull blocks. Both of these alternatives are expensive since they scale as the number of nodes in the entire system. This contrasts to push-based strategies where the maintenance scales as the number of misplaced blocks.

3.2 Local DHash Maintenance

The local maintenance protocol recreates missing fragments on any of the 14 successors of a block's key which do not store a fragment of the block. Each DHash host synchronize its database with each of its 13 successors over the key range from the Chord ID of the host's predecessor up to the host's Chord ID. All keys in this range should be present at the host and its 13 successors. On each Dhash host, the `synchronize()` call will discover any locally missing keys and inform the successors which keys they are missing. For each missing key, `synchronize()` calls `missing()`, which retrieves enough fragments to reconstruct the corresponding block. Finally, `missing()` uses IDA to randomly generate a new fragment.

```
local_maintenance (void)
{
   while (1) {
     foreach (s in mySuccessors[0..12])
       synchronize (s, database[mypredID ... myID])
   }
}

missing (key)
{
   // Called when 'key' does not exist on the host run this code
   block = get (key)
   frag = IDA_get_random_fragment (block)
   merkle_tree.insert (key, frag)
}
```

Figure 3-2: An implementation of the local maintenance protocol. `missing()` is implicitly called by the implementation of `synchronize()`.

The key challenge in the local maintenance protocol is the implementation of `synchronize()`, which is the topic of the next chapter.

Chapter 4

Database Synchronization

Database synchronization compares the fragment databases of two hosts and informs each host which fragments the other host has but which is not locally present. Hosts synchronize a specific range of keys. For example, a host N with predecessor P synchronizes the keys in the range $(P, N]$ with its successors. Other fragments N stores are irrelevant, so they are not included in the synchronization. The database is processed in ranges to make efficient use of local host memory and network bandwidth.

The protocol is designed to optimize for the case in which hosts have most of the fragments for which they are responsible; it quickly identifies which keys are missing with a small amount of network traffic.

4.1 Approach

Synchronization is built out of three components: a database of fragments, an in-core index of the database and a network protocol. The database of fragments is indexed by the fragments' keys, a 20-byte SHA-1 hash.

The index maintains information about which keys are stored in the database. The

index is structured so that two hosts may compare their trees to quickly find any differences between their database. A simplified version of the in-core index is shown below in Figure 4-1.

Figure 4-1: Simplified database indexes for two hosts X and Y.

The database keys are stored in the leaves. The internal nodes hold a SHA-1 hash taken of their children, as in a Merkle tree [29]. For example, the root of the host X holds `3F..D`, which is the hash of `AC..5` and `8E..A`. The root node's left child holds `AC..5`, which is the hash of `03..A` and `7A..3`.

This structure, a recursive tree of hashes, allows two trees to be efficiently compared. The hash value in each internal node is a finger print. If two trees being compared have identical internal nodes, the sub-trees beneath those nodes are identical too and, thus, do not need to be compared.

The algorithm to compare the trees (shown in Figure 4-1) starts by comparing the hash values of the trees' root nodes. In our example, these hash values differ, so the algorithm recurses down the trees, first comparing the left child of each root node. In both trees, these nodes hold the hash value `AC..5`, so the algorithm skips them. The algorithm then compares the the right child of each root node. Here it finds the difference between the two tree. The host X is missing key `F3..4` and host Y is missing key `8E..A`.

The comparison algorithm identifies only the missing keys. It is left to the hosts to

fetch the keys which they are missing. Since the indexes are held in-core on separate DHash hosts, a network protocol is used to send the tree nodes from one host to the other.

The index structure presented in this section is a simplification of the index structure we use. The enhancements are: the index structure does not store the database keys in the tree, and it has a branching factor of 64, not 2.

The following sections specify in detail the fragment database, the database index and the network protocol used to implement database synchronization.

4.2 Database Properties

The database stores and organizes opaque, variable-sized data fragments. Each fragment is identified by a unique 20-byte key. The keys are interpreted throughout the remainder of this paper as 20-byte unsigned integers, ranging in value from 0 to $2^{160} - 1$, inclusive.

The database must support random insertion and lookup efficiently, though removes need not be especially efficient. The algorithms presented below also assume that accessing keys sequentially by increasing value is efficient. For example, when iterating through the database keys, ideally, many keys are read at once, so that disk activity is minimized.

4.3 Database Index Format

Each node maintains a tree in main memory that summarizes the set of keys it holds fragments for; the tree is a specialization of a Merkle tree [29]. Each node in the tree holds a summary of a particular range of the key space, and that node's children divide that range into 64 equal-sized subranges. For example, the root of the tree summarizes the entire key space: $[0, 2^{160})$, and the root's ith child summarizes the range $[i \times 2^{160}/64,\ (i+1) \times 2^{160}/64)$.

Each node n in the Merkle tree is a triple of $\langle hash, count, children \rangle$. The *count* field records the number of database keys within n's range. In an interior node, the *children* field is a pointer to 64-element array of child nodes and NULL in a leaf node. The *hash* field of an internal node is the the hash over its children's hashes:

$$\text{SHA} - 1(child[0].hash \circ ... \circ child[63].hash)$$

The *hash* field of a leaf node is the hash over the concatenation the keys of known fragments in the range of the node:

$$\text{SHA} - 1(key_0 \circ key_1 \circ ... \circ key_m)$$

The fragments that a leaf node refers to are stored in the main fragment database, which is implemented as a B-Tree sorted by key.

A leaf node is converted to an internal node when its *count* field exceeds 64. Ranges with lesser counts are not further subdivided, because such fine granularity information is not needed and would, in fact, consume excessive memory.

The range of a node is not explicitly represented in the node. A node's location in this tree implicately gives its range. For instance, the root node's range is $[0, 2^{160})$ by virtue of it being located of the top of the tree.

This structure is a tree of hashes, which was first proposed by Merkle [29]. Without this hierarchical technique, each node's hash would be equal to the hash over all the the database keys in its range. The root node, for example, would need to rehash the entire database each time a key was added. This is not compatible with the goal of reasonable disk usage.

Since SHA-1 is a cryptographic hash function, the hash is a considered a finger print. If equivalent ranges in two separate Merkle trees have identical hash values, we can assume the two databases are identical in that range. This property will form the basis of the synchronization algorithm present later in Section 4.4.

4.3.1 Index Insertion

As show in Figure 4-2, the insertion routine recursively descends down the Merkle tree starting at the root, guided by the fragment's key. Each successive 6 bits of the key determine which branch to descend down. For example, the top six bits determine which of the $2^6 = 64$ children of the root to descend down. The next six bits determine which of that nodes 64 children to descend down, and so on.

When the insertion routine encounters a leaf node, it inserts the key into the database and returns. On the return path, the insertion routine updates the counts and hashes of all nodes on the return recursion path.

If the insertion routine encounters a leaf nodes that already has a count value of 64, that node is split, by the call to `LEAF2INTERNAL()`, into an internal node with 64 children. This design maintains the invariant that a the database contains at most 64 keys within the range of a leaf node.

The bits of the key determine the precise path that the insertion routine follows. Thus the count and hash values only along the insert path change. Consequently, if a key is inserted into one of two identical databases, their indexes will be identical except along the path where extra key was inserted. This property is crucial to efficient synchronization.

```
insert (fragment f)
{
  // hash values are 160 bits long
  // start with the most siginificant bit
  insert_helper (f, 160, root_node)
}


insert_helper (fragment f, uint bitpos, node n)
{
  // split leaves exceeding 64
  if (ISLEAF(n) && n.count == 64)
    LEAF2INTERNAL (n);

  if (ISLEAF(n)) {
    // recursion bottoms out here
    database.insert (f);
  } else {
    // recurse: next 6 bits of the key select which child
    childnum = (f.key >> (bitpos - 6)) & 0x3f;
    insert_helper (f, bitpos - 6, n.child[childnum]);
  }

  // update count and hash along the recursion return path
  n.count += 1;
  REHASH (n);
}
```

Figure 4-2: An implementation of the Merkle tree insert function.

4.4 Network Protocol

The algorithm used to compare two Merkle tree starts at the root and descends down the trees compare the nodes in each tree. However, the Merkle tree indexes are held in-core on separate DHash hosts. To perform the node comparisons, the algorithm uses the following network protocol to send tree nodes and database keys from one host to the other. The network protocol has two RPCs:

XCHNGNODE Sends a node of the Merkle tree to a remote host. In the reply, the remote host returns the node that is at the equivalent position in its Merkle tree. If

the node is an internal node, the hashes of all children of the node are included. If the node is a leaf, the keys in the database that lie within the range of the node are returned.

GETKEYS Specifies a range and requests that the remote host returns the set of keys in that range in its database. The remote host will return up to a maximum of 64 keys, so as to limit the response size. The client must repeatedly issue the request to get all the keys.

Figure 4-3 shows the code for both the client and the server side of the synchronization protocol. Each host in the system implements both sides of the protocol.

Consider host X initiating a synchronization with host Y. X starts by calling `synchronize()`, which exchanges the root nodes of both X's and Y's trees.

Then, in `synchronize_helper()`, X checks if its node and the node returned by Y are internal nodes. If so, X recurses down the Merkle trees if child hashes differ. X ignores any children that lie outside the key range being synchronized.

In this protocol, X is responsible for walking the Merkle trees if hash values differ, but since X exchanges nodes with Y, both X and Y have the same information. Both have copies of each other's nodes along the paths in their Merkle trees which have differences. This property allows both X and Y to detect the keys each is missing.

Both X and Y call `compare_nodes()` to compare a node in their local Merkle trees to the equivalent node in the remote host's Merkle tree. On both hosts, `compare_nodes()` calls `missing()` to reconcile any differences, by fetching the missing key. In `compare_nodes()`, if the remote node is a leaf, then it contains a list of all the keys stored in the remote host's database in the range of the node. Any of these keys not in the local host's database are passed to the `missing()` function. However, if the local node is a leaf and the remote node

is an internal node, the local host must fetch all the keys in the range of the nodes from the remote host with `GETKEYS`, likely calling `GETKEYS` several times if there are many keys. The local host passes any keys to `missing()` that it does not have in its fragment database.

```
compare_nodes (rnode, lnode)
{
  if (ISLEAF(rnode))
     foreach k in KEYS(rnode)
        if (r.overlaps (k) && !database.lookup (k))
           missing (k)
  else if (ISLEAF(lnode))
     rpc_reply = sendRPC (h, {GETKEYS range=RANGE(lnode)})
     foreach k in KEYS(rpc_reply)
        missing (k)
}

//////////////////// server side ///////////////////////
handle_RPC_request (req)
{
  if (req is {XCHNGNODE node=rnode})
     lnode = merkle_tree.lookup (rnode)
     compare_nodes (rnode, lnode)
     sendRPCresponse (lnode)
  else if (req is {GETKEYS range=r})
     keys = database.getkeys_inrange (r)
     sendRPCresponse (keys)
}

//////////////////// client side ///////////////////////
synchronize_helper (host h, range r, node lnode)
{
  rpc_reply = sendRPC (h, {XCHNGNODE node=lnode})
  rnode = rpc_reply.node
  compare_nodes (lnode, rnode)
  if (!ISLEAF (rnode) && !ISLEAF (lnode))
    for i 0..63
       if r.overlaps(RANGE(lnode.child[i]))
          if lnode.child[i].hash != rnode.child[i].hash
             synchronize_helper (h, r, lnode.child[i])
}

sychronize (host h, range r)
{
  synchronize_helper (h, r, merkle_tree_root)
}
```

Figure 4-3: An implementation of the synchronization protocol.

Chapter 5

Evaluation

The experiments described below use a DHash system deployed on the RON [4] and PlanetLab [1] testbeds. Seventy-seven PlanetLab nodes and eighteen RON nodes are used for a total of ninety-five hosts. 39 nodes are on the East Coast, 30 nodes on the West Coast, 15 nodes in the middle of North America, 9 nodes in Europe and 2 nodes in Australia. Some experiments use only 66 of these nodes.

Most PlanetLab and RON hosts run three independent DHash servers. Three of the RON nodes have lower-bandwidth links (DSL or cable-modem), and run a single DHash server. The total size of the system is 270 DHash nodes.

5.1 Basic Performance

To gauge the basic performance of the DHash system we fetched 2 MB of 8KB blocks from clients at four representative sites: New York University (NYU), Univ. of California at Berkeley, a poorly connected east-coast commercial site (Mazu), and a site in Australia. We also measured the throughput achieved by a long TCP connection from each of the four

client sites to each other node in the test-bed. Figure 5-1 shows the results.[1]

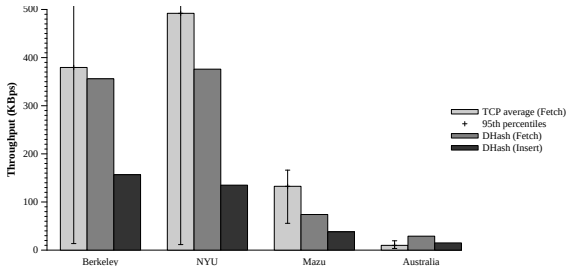


Figure 5-1: Comparison of bandwidths obtained by DHash and by direct TCP connections. Light gray bars indicate the median bandwidth obtained by TCP connections from the site designated by the axis label to each other PlanetLab+RON site. Gray bars indicate the performance of large DHash fetches initiated at the named sites. Black bars indicate the performance of large DHash inserts. As we expect DHash performs more poorly than the best TCP connection, but much better than the worst.

DHash limits its fetch rate to avoid swamping the slowest sender so it does not perform nearly as well as the fastest direct TCP connection. Since DHash fetches from multiple senders in parallel, it is much faster than the slowest direct TCP connection. In general, we expect DHash to fetch at roughly the speed of the slowest sender multiplied by the number of senders. DHash obtains reasonable throughput despite the long latency of a single block fetch because it overlaps fragment fetches with lookups, and keeps many lookups outstanding without damaging the network. DHash inserts slower than it fetches because an insert involves twice as many fragments and cannot use server selection.

5.2 Fetch Latency

While we are mainly interested in throughput, the optimizations presented here also improve the latency of an individual block fetch. Figure 5-2 shows the latency of a block fetch with different sets of optimizations. Each bar represents the average of 384 fetches of different

[1]These results are naturally sensitive to other traffic on the links being measured. For example, the figure shows measurements to Australia during the evening in the US, which corresponds to the higher-demand day time in Australia. By comparison, tests performed earlier in the day showed a fetch and store bandwidth results on the order of 200 KBps.

blocks, performed one at a time. The dark part of each bar indicates the average time to perform the Chord lookup, and the light part of each bar indicates the time required by DHash to fetch the fragments.

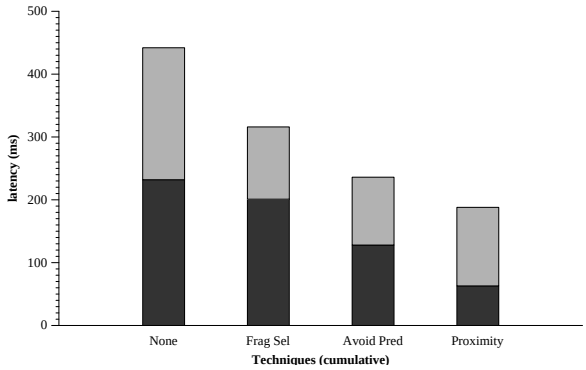


Figure 5-2: The effect of various optimization techniques on the time required to fetch an 8K block using DHash.

The unoptimized system spends roughly an equal amount of time performing lookups as it does fetching the fragments that make up a block. Server selection using the synthetic coordinates is not used.

Adding fragment selection reduces the amount of time required to fetch the block by fetching fragments from the 7 nodes (of the 14 with fragments) closest to the initiator. This effect of this optimization is visible in the reduced size of the fetch time in the second bar (labeled 'Frag Sel'). The lookup time is unchanged by this optimization (the slight variation is likely experimental noise).

The third bar adds an optimization that terminates the Chord lookup on a node close to the target key in ID space, but also close to the originating node in latency. This optimization reduces the lookup time while leaving the fragment fetch time unchanged. It shows the importance of avoiding being forced to contact a specific node during the last stages of a lookup.

The final bar shows the effect using proximity to optimize the Chord lookup. Proximity

routing reduces the lookup component of the fetch time by allowing the search to remain near the initiating node in network space even as it approaches the target key in ID space. Remaining near the initiating node (as opposed to simply taking short hops at each step) is especially important to DHash since it performs lookups iteratively: that is, the initiating node itself contacts each node on the lookup path. The use of coordinates helps to ensure that each node on the path is actually close to the initiator, rather than simply guaranteeing that each hop is short.

Taken together, these optimizations cause a block fetch to complete in 43 percent of the time required by the unmodified system.

5.3 Synchronization Dynamics

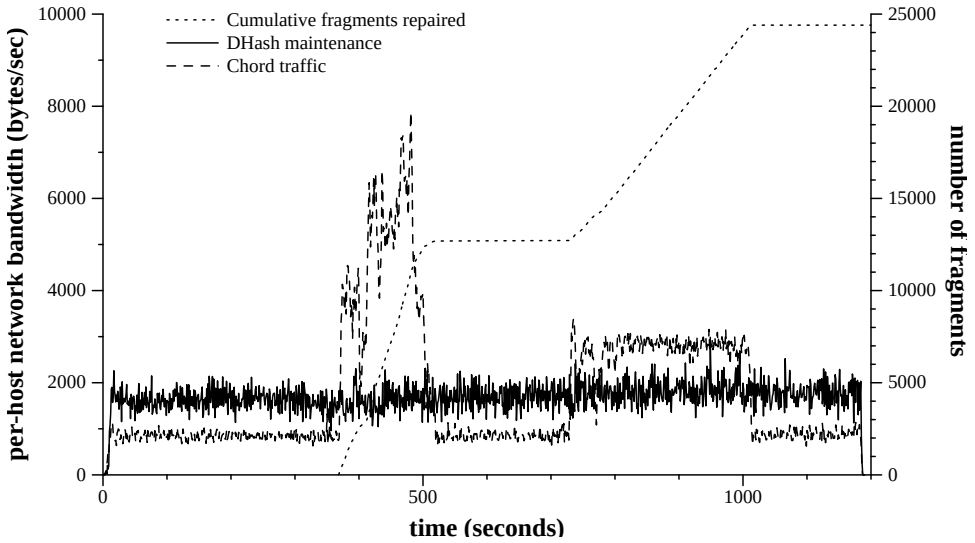


Figure 5-3: A time series graph showing a host leave and then re-join the system.

In this experiment, the system contains 66 PlanetLab hosts each running one server. The RON hosts are not included in this test. The system contains 65536 blocks each of size 8192 bytes, which amounts to 0.5 GB of unique data and 1 GB of total data after erasure encoding.

Figure 5-3 shows the effects of a host leaving the system (at time 380) and then re-joining the system (at time 710). The bandwidth of Chord traffic and the DHash maintenance traffic are plotted over time. The dotted line on the graph plots the cumulative number of fragment repairs.

For the first 380 seconds of the trace, Chord stabilization generates approximately 900 bytes/second of traffic per host in the system, and the DHash maintenance protocols generate around 1700 bytes/second per host. Chord stabilization updates one finger table entry and its successor list ever other second, approximately.

At time 380, a node holding 12688 fragments leaves the system. At this point, the DHash maintenance protocols take effect to restore DHash to its ideal state.

From time 380 until 500 the dotted line, plotting the number of fragment repairs, rises until it reaches 12688 when DHash is again in the ideal state. During this time interval, Chord traffic also increases because of the extra `lookups()` used by the DHash maintenance.

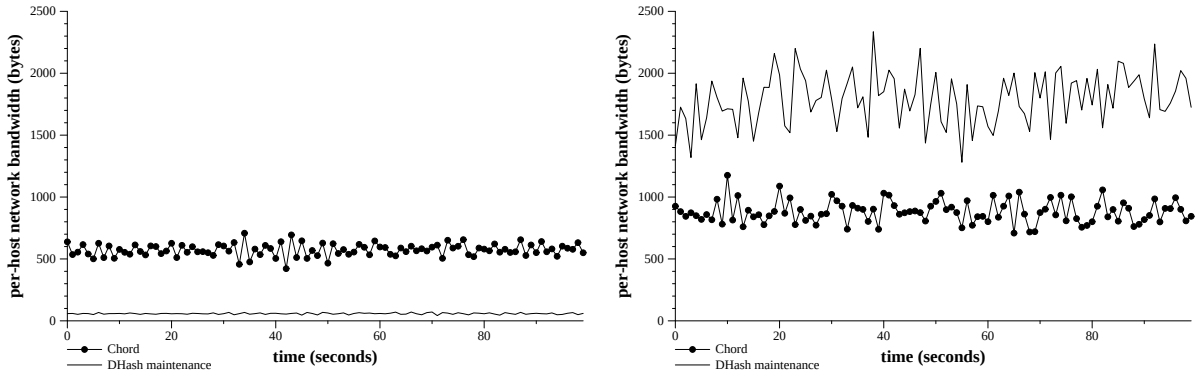
From time 500 until 710, the system is again running Chord stabilization and DHash maintenance at 900 bytes/sec and 1700 bytes/sec per host, respectively.

At time 710, the host rejoins the system with an empty database. From time 710 to 1000, the cumulative fragment repairs line climb as the re-joined host discovers all the fragments it's missing and repairs them. Again, during this time the Chord traffic increases due to extra `lookup()` calls.

After time 1000, the system is again stable and returns to normal operation.

5.4 Ideal State Maintenance

The two graphs in Figure 5-4 show the network traffic consumed by the system when DHash is in the ideal state as defined in Chapter 3. The system contains sixty-six hosts each running



(a) (b)

Figure 5-4: The traffic required when DHash is in the ideal state. Graph (a) shows a system containing no data. Graph (b) shows a system with 0.5 GB of unique data.

one server. The RON hosts are not included in these tests.

In the graph on the left, the database on each host is empty. In the graph on the right, the database is the same as in Section 5.3.

The graph on the right shows around 1700 bytes/host of DHash maintenance traffic. Even though DHash is in the ideal state, hosts cannot just exchange root nodes of the Merkle tree to synchronize. The two hosts store slightly different ranges of keys and as a result their root nodes are not the same. The hosts must walk down their trees to reach the exact key range of the synchronization.

The Chord bandwidth increases from graph (a) to graph (b) because of `lookup()` traffic generated by the DHash global maintenance protocol. If the databases are empty, the protocol performs no Chord `lookup()`s, while if the database is non-empty, the protocol must perform `lookup()`s to verify the fragments in the database of each host are not misplaced.

5.5 Effect of Half-Life

The half-life of a system measures its rate of change. The half-life is the time until half the nodes in system are different, either by new nodes joining or existing nodes leaving [27]. As the half-life grows shorter, the system must do increasing amounts work to keep the system up-to-date with respect to membership changes. Figure 5-5 shows the bandwidth consumed by the system over a variety of half lives.

For small half-lives, the bandwidth of the system is dominated by the fragment traffic needed to repair missing fragments and move misplace fragments. The Chord traffic also increases because of the extra `lookup()`'s generated by the DHash maintenance.

For large half-lives, the bandwidth of the system is dominated by the background Chord stabilization traffic and the DHash maintenance traffic.

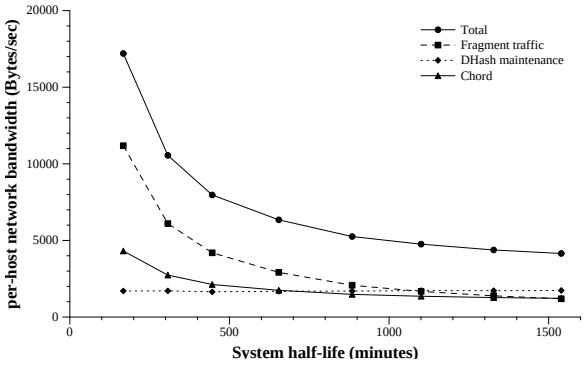
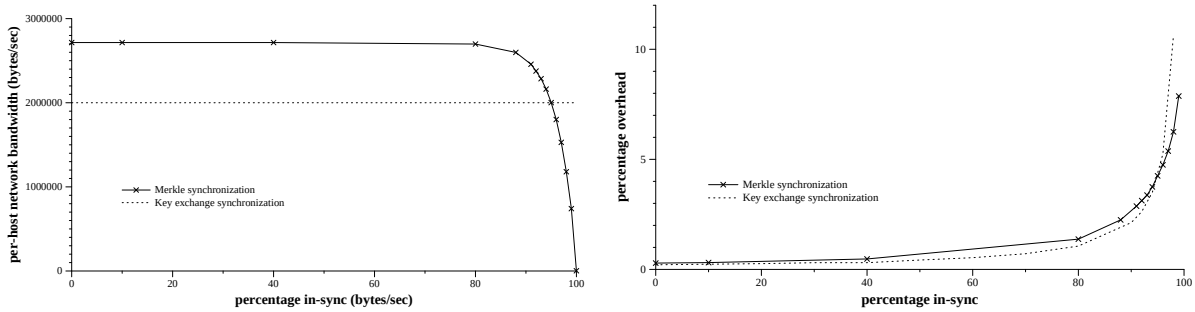


Figure 5-5: Network traffic for various system half lives. The rate of node joins was set equal to the rate of leaves so that the system size averaged 66 DHash hosts. The database setup is the same as in Section 5.3.

5.6 Synchronization Overhead

Figure 5-6 shows the result of an experiment where two hosts synchronized with each other. Each host stored 50,000 fragments. A percentage of these fragments were stored at both hosts and the remaining fragments were unique. The percentage of common fragments is

varied from 0, the hosts have no fragments in common, to 100, the hosts have identical databases.



(a) (b)

Figure 5-6: Sychronization bandwidth (Graph (a)) and overhead with respect to fragment traffic (Graph (b)) between two hosts performing synchronization. Each host stores 50,000 fragments. The percentage of fragments which are identical between the hosts is varied along the x-axis of both graphs. When $x = 0$ the hosts each hold 50,000 unique fragments and when $x = 100$ they each hold the same 50,000 fragments. The dotted line on each graph plots the results for a naive approach to synchronization where hosts synchronize by exchanging their database keys.

The graph on the left plots the total bandwidth consumed by Merkle synchronization when the two hosts synchronize. As a comparison, the graph plots the bandwidth needed to exchange database keys, approximating a naive synchronization algorithm.

As the graph on the left shows, the simple strategy of exchanging keys outperforms Merkle tree synchronization up until the database are 95 % identical. The Merkle tree synchronization sends all the database keys in the `XCHNGNODE` requests and responses, but in addition it exchanges internal tree nodes. Also, the key exchange line does not show RPC overhead.

A DHash host performs synchronization with its successors. If DHash is nearly in its ideal state, these hosts should be nearly identical, except for a very few fragments. Therefore, the important region of the graph is exactly where Merkle tree synchronization uses far less bandwidth than the simple key transfer, above 95 %.

The synchronization protocol uses Merkle trees to prune out large region of the key

space which are identical between the two hosts. The protocol can quickly descend the Merkle tree to find the fragments that are missing.

The graph on the right puts the synchronization bandwidth in perspective by plotting its overhead versus the bandwidth needed to repair a missing fragments. A host must fetch at least 7 other fragments to repair a missing fragment. Here we make a conservative assumption that the fetched fragments are distinct, so that exactly 7 must be fetched. The graph on the right shows that the protocol overhead is in the single digit percentages. The conclusion is that if there is enough bandwidth to transfer the data fragments, then there is certainly enough bandwidth to run the synchronization protocol.

5.7 Memory Usage

The Merkle tree database index was specifically designed to have a small memory foot print. Figure 5-7 measures it size against a fragment database of various sizes. As a point of comparison, the figures also plots the memory consumed by the database keys. This test is run with blocks of 8 KB, which produce fragments of 1170 bytes. The keys are 20-byte SHA-1 hashes.

As the figure shows, even for databases as large as 10 GB, the Merkle tree fits within 10 MB of memory, while the keys exceed 175 MB. The Merkle tree fits well within the memory size of modern desktop computers, while the database keys pose a taxing memory burden. These numbers validate the design decision not to hold the database keys in the Merkle tree. The Merkle tree creates a leaf node, containing just one hash value, instead of holding up to 64 database keys in memory.

The Merkle tree's memory consumption grows as a series of steps. Each step corresponds to the Merkle tree gaining an extra level of internal nodes. The extra levels are added each

time the number database keys reaches approximately 64^n, depending on the exact key values.

Around .3 GB the graph shows a step. This corresponds to 64^3 keys ($64^3 * 1170/2^{20} = 292.5MB$). The steps at 64 and 64^2 are to small to see. The next step, though it is not shown, would occur at $64^4 * 1170/2^{30}$, which is roughly 18 GB.

The Merkle tree's memory consumption is modest even up to large database, 10 GB. However, if necessary, more than one DHash server can be run a physical host in order to make each server's database smaller.

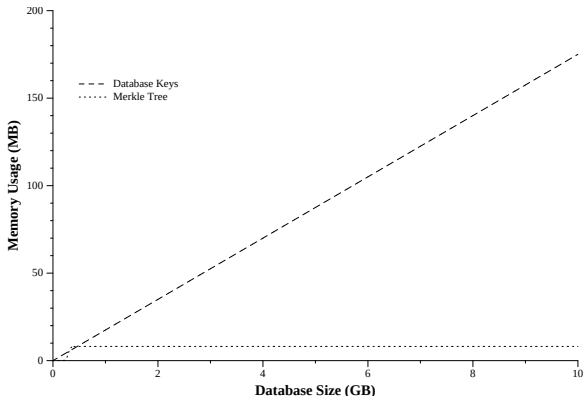


Figure 5-7: Comparison of memory consumed by a Merkle tree and the 20-byte database keys. Both are plot against the database size. Each fragment in the database has a size of 1170, which is the size of fragments for an 8 KB block.

5.8 Fault Tolerance

To examine block availability, we simulated a DHash system using node uptime data gathered from PlanetLab. The PlanetLab data was gathered over the course of two months, and consists of attempts to ssh to all the nodes in PlanetLab. The number of nodes varies from approximately 100 to 130.

This simulation assumes three independent DHash servers are running on each node. We assume that nodes that go off-line will still have all their fragments in their databases

if they rejoin the system. This simulation does not model the DHash local maintenance protocol, which generates missing fragments when nodes go off-line. This simplification makes the simulation pessimistic about block availability because it predicts failure even in cases when there would be enough time to generate missing fragments.

Using the PlanetLab data, with 14 fragments created for every block and seven needed for reconstruction, the simulation indicates that all the blocks are available from DHash greater than 99.8% of the time. Reducing the number of fragments stored in the system greatly reduces the availability of blocks. For example, if four fragments are stored for every block, with two needed for reconstruction, then all the blocks are available only 84.1% of the time. Based on the simulation data, using seven fragments for reconstruction is reasonable on PlanetLab.

Chapter 6

Related Work

DHash closely resembles a number of other storage systems in spirit. The systems include the original replication-based DHash [15, 38], Tapestry/Pond [33, 23], and Pastry/PAST [35, 8]. These systems store data in a DHT-like organization, aim to provide high reliability, and efficiency by exploiting proximity. DHash differs from these systems in its implementation approach: DHash's implementation techniques (proximity routing, server selection, congestion control algorithm) are based on a synthetic coordinate system and a protocol for efficient synchronization of data. The advantages of using the synthetic coordinate system are (1) proximity routing, server selection, and congestion control becomes simple and (2) reduces communication traffic, because nodes don't have to be probed.

A number of other systems use DHT-like organizations to store mappings from key to location instead of storing the actual data. Examples includes i3 [37] and Overnet [30], which uses Kademlia [28]. Overnet is in particular interesting because it is deployed on a large scale, but, unfortunately its implementation is closed and little experimental data is available [32].

The Distributed Data Structure system (DDS) provides a hash table interface to data

distributed across a number of machines [21]. Like DHash, DDS replicates data to ensure availability. Unlike DHash, DDS is intended to be run on a tightly connected cluster of servers rather than on the wide-area network.

Although FarSite [3] is not organized as DHT, it replicates files and directories on a large collection of untrusted computers, and provides Byzantine fault tolerance. FarSite performs whole-file replication without the use of coding, and provides stronger security properties than DHash. Making DHash resistant to malicious participants is an area of future research.

6.1 Cooperative Backup

A number of papers have proposed to organize backup in a cooperative fashion. Pastiche uses Pastry to locate its buddies [34]. Cooperative backup uses a central server to find partners [18], but uses coding to spread the blocks. HiveCache is a reincarnation of MojoNations focused on cooperative backup for enterprises [24]. Stronger versions of cooperative backup, namely archiving data forever in the presence censors, also have been proposed: examples include the Eternity Service [5] and FreeNet [12].

6.2 Server Selection and Spreading

Achieving low-latency via server selection is common in the Internet, most well known by its use in content distribution networks (CDNs) such as Akamai, Speedera, Digital Island and Exodus. However, being targeted for the Web means that there can be no client support for selection of the optimal server. Instead, Web CDNs typically rely on DNS redirection. The effectiveness of such techniques depend on the assumption that the client is close to its

name server — this has been shown to often be a poor assumption [36].

File-sharing peer-to-peer systems provide natural replicas because many peers want copies of the same files — some of these systems also use striped file transfers to help peers fetch from the fastest sources. Examples of such systems include BitTorrent [6] and KaZaa [25].

6.3 Coding

A wide variety of storage and network systems use coding to recover lost data. In the context of peer-to-peer systems, Weatherspoon and Kubiatowicz presented a nice argument for using coding instead of replication [40]. Peer-to-peer archival systems in particular have made use of coding or secret sharing schemes to achieve high robustness [11, 5, 26, 22]. Most systems use Reed Solomon or Tornado erasure codes, whereas Mnemosyne [22] and DHash use IDA [31].

6.4 Replica Synchronization

Like coding, efficient synchronization of replicas is also a well-studied topic. Rsync [39] is a widely-used tool for file synchronization that exchanges just the delta between two replicas. The use of Merkle hash [29] trees for efficient synchronization is used by a number of systems. Duchamp uses hierarchical hashes to efficiently compare two file systems in a toolkit for partially-connected operation [17]. BFS uses hierarchical hashes for efficient state transfers between clients and replicas [9, 10]. SFSRO uses Merkle hash trees both for computing digital signatures and synchronizing replicas through incremental updates efficiently [20].

Chapter 7

Summary

7.1 Conclusions

This thesis presented the DHash distributed hash table and the mechanisms it uses to provide robust and efficient operation. The design combined five techniques in a novel way. The techniques include erasure coding, replica synchronization, synthetic coordinates, proximity routing, and server selection to provided robust and efficient operation.

The design centers around a storage representation which stores each block as a set of erasure encoded fragments. The fragment maintenance protocols restore any destroyed or misplaced fragments caused by system membership changes. The fragments combine with the synthetic coordinates to provide high-throughput and low-latency block fetch and store.

7.2 Future Work

More optimizations remain for DHash, particularly for its maintenance protocols. Currently DHash takes an eager approach to maintenance: as soon as a host disconnects the fragments it stores are recreated on other hosts. If DHash recognized the difference between a tempo-

rary host disconnection and a permanent host departure from the system, it could take a lazy approach [32, 7], only recreate missing fragments, if it knew the host was permanently departed. This optimization removes a lot of needless fragment maintenance that occurs when a host disconnects and later reconnects.

One approach to approximate this behavior is to redefine DHash's *ideal state* so that within the 16 successors of a block's key at least some minimum number, such as 12, distinct fragments must exist. Hosts would only recreate fragments if the total number of fragments for a block dips below this minimum value. In this new design, each host would need to count the number of fragments for a block. It could be challenging to compactly represent this state in memory. Moreover, the Merkle tree's of neighboring hosts on the Chord ring will likely be significantly different. The greater number of differences would create more traffic for each synchronization invocation. Also, block fetches might be slower if many fragments are missing.

The DHash synchronization can be optimized. Even when DHash is in the ideal state, hosts must walk down their trees to reach the exact key range of the synchronization. They cannot just exchange root nodes of the Merkle tree to synchronize, since they store different key ranges. However, since each host synchronizes with the same hosts repeatedly, a host can cache results from previous synchronizations to optimize future synchronizations with the same host. Specifically, a host should cache the tree nodes of hosts with which they synchronize. A subsequent synchronization does not need to descend down a tree node if the node received from the remote host matches an already cached node from a previous synchronization.

Bibliography

[1] Planetlab. `http://www.planet-lab.org`.

[2] Project Iris. `http://www.project-iris.net`.

[3] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

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

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

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

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

[8] 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. Short version in International Workshop on Future Directions in Distributed Computing (FuDiCo), Bertinoro, Italy, June, 2002.

[9] Castro, M., and Liskov, B. Practical Byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation* (February 1999), pp. 173–186.

[10] Castro, M., and Liskov, B. Proactive recovery in a Byzantine-fault-tolerant system. In *4th Symposium on Operating Systems Design and Implementation* (2001).

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

[12] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. ICSI Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, California, June 2000). `http://freenet.sourceforge.net`.

[13] Cox, L. P., and Noble, B. D. Pastiche: making backup cheap and easy. In *5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

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

[15] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001). `http://www.pdos.lcs.mit.edu/chord/`.

[16] Dabek, F., Zhao, B., Druschel, P., and Stoica, I. Towards a common api for structured peer-to-peer overlays. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'02)* (Feb. 2003). `http://www.cs.rice.edu/Conferences/IPTPS02/`.

[17] Duchamp, D. A toolkit approach to partially disconnected operation. In *Proc. USENIX 1997 Ann. Technical Conf.* (January 1997), USENIX, pp. 305–318.

[18] Elnikety, S., Lillibridge, M., Burrows, M., and Zwaenepoel, W. Cooperative backup system. In *FAST 2002* (Jan. 2002). WiPs paper; full version to appear in USENIX 2003.

[19] Freedman, M., and Mazieres, D. Sloppy hashing and self-organizing clusters. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'02)* (Feb. 2003). `http://iptps03.cs.berkeley.edu/`.

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

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

[22] Hand, S., and Roscoe, T. Mnemosyne: Peer-to-peer steganographic storage. In *1st International Workshop on Peer-to-Peer Systems (IPTPS'02)* (Mar. 2001). `http://www.cs.rice.edu/Conferences/IPTPS02/`.

[23] Hildrum, K., Kubatowicz, J. D., Rao, S., and Zhao, B. Y. Distributed Object Location in a Dynamic Network. In *Proc. 14th ACM Symp. on Parallel Algorithms and Architectures* (Aug. 2002).

[24] Peer to peer (p2p) enterprise online backups. `http://www.mojonation.net/`.

[25] KaZaa media dekstop. `http://www.kazaa.com/`.

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

[27] Liben-Nowell, D., Balakrishnan, H., and Karger, D. R. Analysis of the evolution of peer-to-peer systems. In *Proc. PODC 2002* (Aug. 2002).

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

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

[30] Overnet. `http://www.overnet.com/`.

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

[32] Ranjita Bhagwan, S. S., and Voelker, G. Understanding availability. In *Proceedings of the 2003 International Workshop on Peer-to-Peer Systems* (Feb. 2003).

[33] Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., and Kubiatowicz, J. Pond: The oceanstore prototype. In *FAST 2003* (Mar. 2003).

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

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

[36] Shaikh, A., Tewari, R., and Agrawal, M. On the effectiveness of DNS-based server selection. In *IEEE INFOCOM* (2001).

[37] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM 2002* (August 2002).

[38] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM* (San Diego, Aug. 2001). An extended version appears in ACM/IEEE Trans. on Networking.

[39] Tridgell, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Apr. 2000.

[40] Weatherspoon, H., and Kubiatowicz, J. D. Erasure coding vs. replication: A quantitative comparison. In *Proc. 1st International Workshop on Peer-to-Peer systems* (Mar. 2002).