A DHT-based Backup System

Emil Sit, Josh Cates, and Russ Cox
MIT Laboratory for Computer Science

10 August 2003

1 Introduction

Distributed hash tables have been proposed as a way to simplify the construction of large-scale distributed applications (e.g. [1, 6]). DHTs are completely decentralized systems that provide block storage on a changing collection of nodes spread throughout the Internet. Each block is identified by a unique key. DHTs spread the load of storing and serving blocks across all of the active nodes and keep the blocks available as nodes join and leave the system.

This paper presents the design and implementation of a cooperative off-site backup system, Venti-DHash. Venti-DHash is based on a DHT infrastructure and is designed to support recovery of data after a disaster by keeping regular snapshots of file systems distributed off-site, on peers on the Internet. Whereas conventional backup systems incur significant equipment costs, manual effort and high administrative overhead, we hope that a distributed backup system can alleviate these problems, making backups easy and feasible. By building this system on top of a DHT, the backup application inherits the properties of the DHT, and serves to evaluate the feasibility of using a DHT to build large scale applications.

The backup system is based around the Venti archival storage system [9], replacing the storage back-end with the DHash distributed hash table [5]. Venti-DHash operates as an archiver that takes complete file system snapshots, at a block level. Each unique block is only stored once, even across snapshots. DHash is used to balance storage and network load, as well as to provide adequate availability blocks.

A number of changes were made the internals of DHash in order to meet our desired performance and availability goals. Our improved version of DHash is a DHT with good read and write performance, and 5 nines of availability per block (assuming an average node reliability of 90%). The resulting system is now being tested by running backups of our primary file server.

The rest of the paper is structured as follows. Section 2 briefly surveys related work. The design of the backup system is presented in Section 3. Next, we describe how DHash was changed to achieve the desired performance and availability goals in Section 4. Section 5 describes some preliminary performance benchmarks and analysis we have conducted on our prototype. Finally, we conclude in Section 6.

2 Related Work

Venti-DHash is based on combining two systems. Venti [9] is an archival storage system, designed for archiving data in a read-only fashion efficiently to hard disks. While Venti itself is application independent, we use the ideas for physical backup presented in [9]. DHash is a distributed hash table built on top of the Chord lookup protocol [5]. Chord organizes nodes into a sorted ring of 160-bit identifiers without central control; DHash stores blocks keyed by 160-bit identifiers on the Chord node that is the successor of the block's key.

Several peer-to-peer derived backup systems have appeared in the past few years. Lillibridge *et al* described a cooperative Internet backup scheme [7]. This system is not based on a DHT — instead, they arrange for introductions between systems that have roughly equal storage availability and requirements via a centralized server. Each peer uses the introducer to find a set of sufficiently diverse peers that will hold its backups. The paper also discusses a number of different attacks and methods needed to defend against these attacks.

Pastiche is a peer-to-peer backup system that relies on Pastry to find buddies that have largely similar content [3]. For example, a machine will seek out a buddy machine running the same operating system, ensuring that the two machines will share a fair amount of system data. However, user home directories are less likely to share content than two machines running the same version of Windows. Our system does not rely on the need to find appropriate partners for the system. Instead, we store encrypted

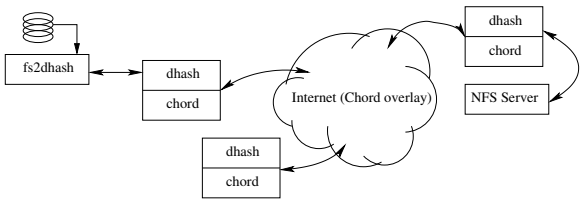

Figure 1: Backup Architecture

blocks out on the system, and use efficient snapshots to minimize the additional overhead on the system.

Some of the improvements to DHash were based on ideas used in the OceanStore prototype [11]. Compared to OceanStore, DHash is simpler and focused more on performance. Features such as Byzantine fault tolerance are not present in DHash.

3 Design

Our backup system must provide a number of general high-level qualities.

1. Availability: It is not acceptable for a block to be lost.
2. Confidentiality: Administrators and users should be able to specify that data is to be encrypted so that random peers storing blocks will not have access to private data.
3. Ease of use: The system should operate relatively transparently to the administrator and end user.
4. Performance: System should be able to store and retrieve large quantities of data at reasonable speeds. In particular, it must be possible to back up a full day's incremental changes to a typical file system in a few hours.

A high-level block diagram of Venti-DHash is shown in Figure 1. The system operates at a filesystem level: periodically, a snapshot is taken of all the active blocks in a filesystem. The blocks are then streamed directly into DHash, organized as a Venti stream, described below. DHash provides availability and performance (as described in Section 4); Venti provides the confidentiality and ease of use.

To provide confidentiality, Venti-DHash builds an encrypted block store on top of the simple `put` and `get` interface provided by DHash. DHash takes a given data block and stores it directly using a 20-byte key $k = \mathrm{SHA1}(\text{block contents})$. Prior to storing disk blocks in DHash, Venti-DHash first encrypts data using AES. Thus, keys in the backup system are 40-byte signatures. The first half specifies the hash used to fetch the encrypted block from DHash: this is the SHA1 of the content of the encrypted block. The second half of the key specifies the cryptographic key that was used to encrypt the block. Selection of the encryption key is discussed below.

Given this encrypted block store primitive, Venti-DHash builds an encrypted stream store primitive that stores an arbitrary stream of data, recursively hashing it down to a 40-byte signature. The recursive hash procedure works conceptually as follows. The stream is prefixed with a header marking it as a data stream. Then the stream is broken into eight kilobyte chunks, each of which stored under its 40-byte signature. The 40-byte signatures for each chunk are concatenated to form a new stream, which is prefixed with a header marking it as a signature stream. The the "break into chunk, hash, and concatenate the signatures" cycle is repeated until a single signature remains, which is the signature for the entire stream. The result is a perfectly balanced tree with the original data stream at the leaves. Such a structure allows for efficient random access. On updates, if some bytes in a file change, only the modified blocks and their ancestors will have different signatures upon rehashing, so the bulk of the tree can be reused in a subsequent store operation.

When time comes to restore, two methods are possible. One application communicates with DHash and streams data out of DHash and creates a FFS image. The image could be directly written to a physical disk device. For interactive use, we can view each snapshot as a read-only file system, using a loop-back NFS server that retrieves data from DHash. By requesting specific blocks out of DHash, the NFS server can reconstruct any snapshot of the backed up filesystem on the fly, and present it as a real browsable filesystem to the user.

Encryption poses an interesting problem: if multiple people insert the same block, we would like not to store it multiple times. To achieve this, we use the SHA1 hash of the unencrypted block as its encryption key. Then if blocks are identical, they will have identical encryption keys and thus identical encryptions (assuming a deterministic encryption function). Such a scheme provides decent confidentiality from casual snooping but is susceptible to a dictionary attack: a block's contents can be guessed and then verified. To avoid dictionary attacks, we could encrypt blocks with a random key instead of the plaintext hash. Pseudocode for the insert operation is:

```
sig
insert(block b, bool private)
{
    if(private)
        key = random();
    else
        key = sha1(b);
```

```
    eb = encrypt(b, key);
    ehash = sha1(eb);
    if(!dhash_lookup(ehash))
        dhash_insert(ehash, eb);
    return concat(ehash, key);
}
```

The price of using a completely random key is that the hash of the encrypted block changes at every backup even when the block has not. It is possible to generate the random key only once per block and store it somewhere on the system. On a per-node basis, this restores the property that each unique block is stored only once. Across peers, identical blocks are encrypted with different random keys and therefore cannot be coalesced.

4 DHash and Chord Improvements

In this section, we describe some of the changes that were made to DHash and Chord in order to achieve the performance and reliability goals that are needed for a robust backup application.

4.1 Erasure codes

In order to improve the reliability of the system, we chose to change DHash to use an erasure code to store data. As has been well established in the literature (e.g. [7, 12]), erasure codes offer significantly improved availability for a given amount of storage overhead. With erasure codes, a block of data is encoded into k fragments, of which some subset m, are needed to reassemble the message. We chose to use Rabin's Information Dispersal Algorithm (IDA) [10] for our erasure code, with $p = 65537$. IDA has the nice property that for a given m, practically any number of fragments can be generated and with probability $1 - \frac{1}{p} = \frac{65536}{65537}$), any m subset of these fragments can be used to reconstruct the original block.

This algorithm has the benefit that it allows anyone with a copy of the block to produce one of a very large set of possible fragments. This allows nodes who wish to acquire a fragment to easily produce a fragment that is distinct from all of the other fragments in the system. This is important for our data synchronization technique described below. However, a major downside is that the current encoding makes it infeasible for servers to verify the validity of individual fragments.

Our system is nominally designed for blocks that are 8K in size, though it can easily accommodate smaller and larger blocks. We divide all blocks into $k = 14$ fragments and store them on the successors of the key: DHash uses Chord to find the successor of the key in the ring and obtain the current successor list. DHash then pushes a unique fragment to the first k successors. An 8K block can easily be fragmented such that $m = 7$ fragments are needed for reassembly. This gives each fragment a size slightly larger than 1K, which makes it easy to fit into a single UDP packet. A value of $m = 7$ also gives a two-times storage overhead for five nines of availability.

4.2 Efficient data synchronization

One key problem in large distributed distributed hash tables is how to efficiently move data between nodes: as nodes enter or leave the network, the data that each node is responsible for changes. Some mechanism must ensure that a sufficient number of distinct fragments remains in the system to guarantee availability, and that these fragments are stored on the correct nodes, so that they can be found. Data must be moved around efficiently and with a minimum amount of overhead.

DHash uses two maintenance procedures to ensure that enough distinct fragments exist in the correct locations. The first is a local procedure that ensures that new nodes joining the successor set for a given block have the fragments that they need (e.g. because a complete new node has joined the system, or because a node in the set has left the system). Nodes in each group of successors continuously exchange summaries of block lists (represented compactly as Merkle trees), so that they can quickly determine whether they are missing fragments for blocks that they are responsible for. When missing fragments are discovered, the node that is missing the fragment reconstructs the block and generates a new fragment.

A global procedure is responsible for moving block fragments that are very far out of place. This can happen when a very large number of nodes joins the system, for example. Each node scans its own database periodically, looking for fragments from blocks that it is not responsible for. For these fragments, the node will perform a lookup for the key and attempt to push each fragment to its proper home.

These algorithms are described in more detail in [2].

4.3 Locality awareness

Chord lookups and DHash retrievals must contact a large number of nodes in the network. Because node IDs are assigned randomly, a lookup and fetch may incur significant latency from having to contact nodes on the other end of trans-oceanic links. Fortunately, data and routing information is replicated and we can attempt to select the closest replica, if the nodes have access to some latency information.

We chose a distributed machine learning algorithm that can quickly synthesize coordinates based on latency measurements and converges relatively quickly [4]. The coordinates place each node in a three dimensional Euclidean space where the distance between each pair of nodes is an estimate of the expected latency between the two nodes. While this algorithm is still undergoing refinement, it can definitely distinguish between nodes in different continents and can often provide real-time estimate of latencies.

Each node starts at a random set of coordinates. Each RPC call includes the current coordinates of the caller and the response includes the RPC current coordinates of the receiver. The caller uses the measured round-trip time to adjust his own coordinates. Over time, each node independently migrates to a location that minimizes the error that the coordinates predict for the latencies that it measures to other nodes.

This coordinate information can be used to improve a number of performance issues in Chord and DHash. For example, by including coordinates of nodes in Chord and DHash messages, a node can immediately estimate the expected round trip time to one of the nodes included in the message, even if it has never communicated with that node before. This allows the transport protocol to correctly detect RPC timeouts. Because timeouts indicate failures, reducing the number of incorrect timeouts helps keep routing tables stable.

For DHash, coordinates allow clients to select fragments to download from optimal peers. When the successor list for a block is retrieved, the client is given the coordinates for each of the successors. Since only m of the more than k successors need to be contacted, DHash clients can sort the successor list and select the m most optimal peers.

5 Evaluation

We have implemented this system for Unix systems, such as FreeBSD, running the Berkeley Fast File System (FFS). The DHash/Chord software is written in C++, using the event-driven UDP RPC package provided by the SFS toolkit [8]. The Venti archiving portion was written in C and largely ported from the Plan 9 implementation of Venti.

5.1 Performance

We have tested the Venti-DHash prototype by storing data onto a Chord network running on PlanetLab and RON.

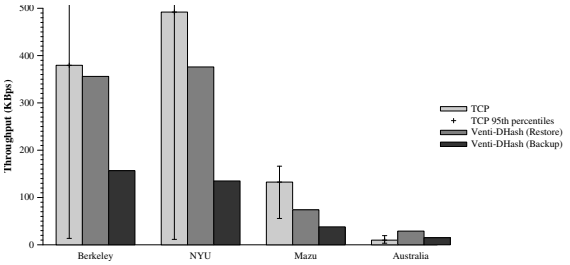

Figure 2: Comparison of bandwidths obtained by Venti-DHash and by direct TCP connections from several different PlanetLab+RON sites.

Figure 2 shows the read and write performance when running on a testbed of seventy-seven PlanetLab nodes and eighteen RON nodes. As we expect Venti-DHash performs more poorly than the best TCP connection, but much better than the worst.

5.2 Usability

Our implementation also attempts to make the system easy to use, though some extra administrative overhead remains. A single binary provides access to the enhanced DHash DHT: the administrator must configure a machine (or set of machines) that will provide DHT service, contributing disk space to the system and providing a gateway for local backups. This can be a central service provided by a site administrator. On each workstation to be backed up, a lightweight client application runs nightly, processing raw disk partitions for changes and storing new blocks, encrypted and fragmented, into DHash via an RPC protocol to the local DHash gateway. The loop-back NFS server can be set to automatically make each daily snapshot available as it becomes available.

6 Conclusion and Future Work

Peer-to-peer backup systems show significant potential for enabling cheap and easy backup. We have built a cooperative backup application that uses a distributed hash table infrastructure to simplify the application's complexity. We found that certain aspects of our DHT implementation needed improvement, but we were able to address the most important issues using erasure coding and locality awareness.

Of course, there are aspects of the system that we hope to improve. For example, we would like to improve the verifiability of fragments in DHash so that servers can

avoid simple disk space wasting attacks. Fortunately, because Venti-DHash is largely independent of DHash, any subsequent improvements to DHash will be available to the system as well. We expect usability to improve with time as we gain experience with the system.

In the future, we hope to explore how DHTs such as DHash can be used to build other large-scale distributed applications and how the needs of those applications will interact with the design and features of DHTs.

References

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

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

[3] COX, L. P., AND NOBLE, B. D. Pastiche: making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[4] COX, R., AND DABEK, F. Learning Euclidean coordinates for internet hosts. `http://www.pdos.lcs.mit.edu/~rsc/6867.pdf`, Dec. 2002.

[5] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001). `http://www.pdos.lcs.mit.edu/chord/`.

[6] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, Nov. 2000), pp. 190–201.

[7] LILLIBRIDGE, M., ELNIKETY, S., BIRRELL, A., BURROWS, M., AND ISARD, M. A cooperative backup system scheme. In *Proceedings of the 2003 USENIX Technical Conference* (June 2003).

[8] MAZIÈRES, D. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference* (June 2001), pp. 261–274.

[9] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)* (2002).

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

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

[12] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002).