Defragmenting DHT-based Distributed File Systems

Jeffrey Pang[*] **Phillip B. Gibbons**[†]
Michael Kaminsky[†] **Srinivasan Seshan**[*]
Haifeng Yu[‡]

March 2007
CMU-CS-07-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Computer Science Department, School of Computer Science, Carnegie Mellon University
[†]Intel Research Pittsburgh
[‡]Computer Science Department, School of Computing, National University of Singapore

This work is supported by NSF grant CNS-0435382 and NUS grants R-252-050-284-101 and R-252-050-284-133.

Keywords: DHT, locality, defragmenting, file systems

Abstract

Existing DHT-based file systems use consistent hashing to assign file blocks to random machines. As a result, a user task accessing an entire file or multiple files needs to retrieve blocks from many different machines. This paper demonstrates that significant availability and performance gains can be achieved if, instead, users are able to retrieve all the data needed for a given task from only a few DHT nodes. We explore the design and implications of such a "defragmented" DHT-based distributed file system, called D2, that also maintains important DHT properties like storage load balance. We show using real-world file system traces that a simple key encoding scheme is sufficient to maintain good defragmentation for most user tasks. Using both simulation and an actual 1,000 node deployment, we show that D2 increases availability by over an order of magnitude and improves user-perceived latency by 30–100% compared to a traditional design.

1 Introduction

Distributed Hash Tables (DHTs) enable self-managing file systems that can aggregate the storage of thousands or millions of nodes. In a DHT-based file system, an arbitrarily large number of nodes can self-organize to provide efficient data location, high data availability, and storage load balance. There are many approaches that achieve efficient data location and high availability for individual objects, but nearly all DHTs [3, 8, 11, 32, 36] rely on *consistent hashing* [22] or a close variant to balance storage load. In consistent hashing, each node has a random *ID*, and each data object is hashed to obtain a *key*. The object is assigned to the node whose ID is the immediate successor of its key (i.e., the smallest ID that is larger than its key). This process ensures that objects are distributed close to uniformly across all DHT nodes, hence balancing load.

Unfortunately, because assignment based on hashing is random, an immediate result of this approach is that related objects are spread across many nodes. For example, in CFS [8], where each file block has a distinct key, each block accessed by a user or application task is most likely assigned to a different node. Even if we treat entire files as variable-size data objects, as in PAST [38], a user or application may access multiple files to complete a task. This "fragmentation" has two consequences. First, many tasks will fail if any of the objects it requires is unavailable (e.g., compiling a large project). Therefore, the likelihood that a task fails is greater than if the objects were less fragmented. Second, a new DHT lookup must be performed to locate each block accessed by the task since each block is likely to be stored on a different node. Since the latency of a single lookup can be several RTTs, these lookups can cause a task's completion time to be much larger than the time it takes to download the objects alone. In addition, this large volume of DHT lookup traffic limits the bandwidth that nodes have left for optimization and maintenance, hurting performance and robustness.ereafter, we refer consistent hashing DHTs with block objects as *traditional* DHTs and consistent hashing DHTs with file objects as *traditional-file* DHTs.

In this paper, we explore the design of a "defragmented" DHT-based file system that preserves data locality when assigning objects to nodes. In this design, we address three principle challenges:

1. Maintaining the locality of objects accessed by arbitrary tasks may not be possible without future knowledge; for example, tasks may access objects randomly. Is the structure inherent in file system tasks sufficient to preserve locality in a DHT without foreknowledge and undue complexity?

2. There is a trade-off between preserving data locality and the amount of parallelism users can exploit when fetching data. Data locality can be leveraged to reduce other overheads, but are these reductions enough to offset this cost?

3. Object keys are no longer distributed uniformly in the key space when data locality is maintained. Hence, if nodes are responsible for roughly equal ranges of the key space, as in a traditional DHT, storage load would not be balanced. However, load balance is necessary to limit both the maximum storage that each node must provision and the worst case data regeneration cost incurred upon failures.an load still be balanced without significant overhead?

To address these questions, we implement and evaluate a prototype called D2 (*D*efragmented *D*HT) as a concrete case study. We make three primary design contributions to affirmatively answer each respective question:

Locality Preserving Keys. Using real file system traces and web traces [16, 12, 17], we demonstrate that choosing keys based on *name-space* locality (i.e., the ordering of files as dictated by a preorder traversal of the directory hierarchy) is close to optimal in terms of minimizing the number of nodes that each user needs to access to fetch data.

Lookup Caches. D2 leverages the locality in keys by caching the key ranges of nodes in lookup results. By using these caches, D2 reduces total DHT lookup traffic by up to 95% compared to traditional DHTs. In a large DHT system with geographically disperse nodes, the effect this reduction has on end-to-end latency more than outweighs the cost of more limited parallelism for real file system tasks.

Load Balancing Overhead. D2 uses existing active load balancing techniques (originally designed for range queries) to balance storage load [4, 21, 23]. The cost of active load balancing is data migration traffic, so D2 uses *pointers* to minimize unnecessary data migration. When using pointers, the migration traffic required to maintain load balance is about 50% of the write traffic in a real file system workload. In a web caching workload with much more significant churn in its data distribution, migration traffic is still comparable to the write traffic. For a typical file system where read traffic dominates write traffic, this overhead is not large relative to the volume of other data transfers.

In addition to showing the value of these techniques in a defragmented DHT-based file system, we conduct an extensive evaluation on Emulab [39] with up to 1,000 instances of our prototype and with long term simulations. D2 decreases the failure rate of user tasks in a real file system workload by over an order of magnitude. Moreover, D2 improves user-perceived latency by 30% to 100% in a 1,000 node system.

This paper is organized as follows: Section 2 describes related work. Section 3 provides an overview of D2. Sections 4, 5, and 6 describe D2's unique design aspects, and Section 7 describes its implementation. Sections 8, 9, and 10 present evaluations of D2's availability, performance, and load balancing overhead. Section 11 concludes.

2 Related Work

Clustering and Defragmentation. Clustering related data [10, 28, 29] and defragmentation [19] are well known techniques for maintaining data locality on disks, which improves the performance of local file systems. The idea of clustering related blocks and files in distributed file systems is also not new — the Andrew File System [15] clusters files into volumes to improve system operability and Archipelago [20] clusters the files in a directory on the same node to provide fault isolation. However, these file systems require manual configuration and management. This paper generalizes the clustering and defragmentation concepts to DHT-based file systems to take advantage of their existing self-management and scalability properties.

Caching. An orthogonal but related concept used to improve the performance and availability of distributed file systems is caching. For example, AFS clients cache whole files to avoid contacting file servers for every operation. Coda [24] clients cache files so that users can operate on them even while file servers are unavailable. At one extreme, a client could store the primary copy of data itself, avoiding file server communication and unavailability entirely. However, files are often shared and accessed at multiple locations. Hence, there is a trade off between caching and consistency. Like most file system designs, we value long term consistency, so D2 clients only cache data for short time periods. Other points in this design space, although interesting, are outside the scope of this paper.

DHTs and Availability. Numerous DHT-based file systems have been proposed, including CFS [8], Ivy [32], Pond [36], PAST [38], Total Recall [3], and Glacier [11]. All of these systems use consistent hashing (or a variant) to balance load. Some of these systems are designed to improve the availability of *individual* objects [3, 6, 11] (e.g., using erasure coding and active availability monitoring). However, these techniques only work well when the the granularity of object access (e.g., block, file, or directory) is known beforehand — for example, when applications only require access to individual files to complete tasks. Often this granularity is not known when objects are created or is too cumbersome for users to specify. This is especially true for legacy applications that access DHTs using a traditional file system interface. Defragmentation does not preclude the use of these techniques, so they can be used in conjunction with D2's locality preserving data assignment.

The availability of multi-object tasks is also studied by Yu *et al.* [40]. However, Yu *et al.* mainly focus on the availability difference resulting from object correlation when the fragmentation level is fixed. In comparison, we investigate how to defragment related objects for better availability and performance in the context of a decentralized file system.

Administrative Locality. Mislove *et al.* [31] and Harvey *et al.* [13] propose DHTs that maintain administrative locality — i.e., their systems can constrain the set of nodes on which each object is placed. Although both these systems can be used to preserve data locality, neither can automatically maintain global load balance. Harvey *et al.*'s DHT can balance load within a particular section of the DHT (e.g., an administrative domain) using consistent hashing, but does so at the cost of data locality. D2's defragmentation techniques can complement these systems by maintaining data locality and load balance within each administrative domain.

Active Load Balancing. DHTs designed to support range queries [4, 21] pioneered the dynamic load balancing algorithm used in D2. Our primary contributions in this area are the application of dynamic load balancing for preserving file system locality and the evaluation of the algorithm's stability and overhead under real file system workloads.

3 D2 Overview

To place our specific contributions in context, this section provides an overview of D2's architecture. Design details unique to D2 are discussed in the subsequent sections.

Usage Assumptions. Like other DHT storage systems, D2 uses the storage and bandwidth resources contributed by a large community of nodes. We make the same usage assumptions as

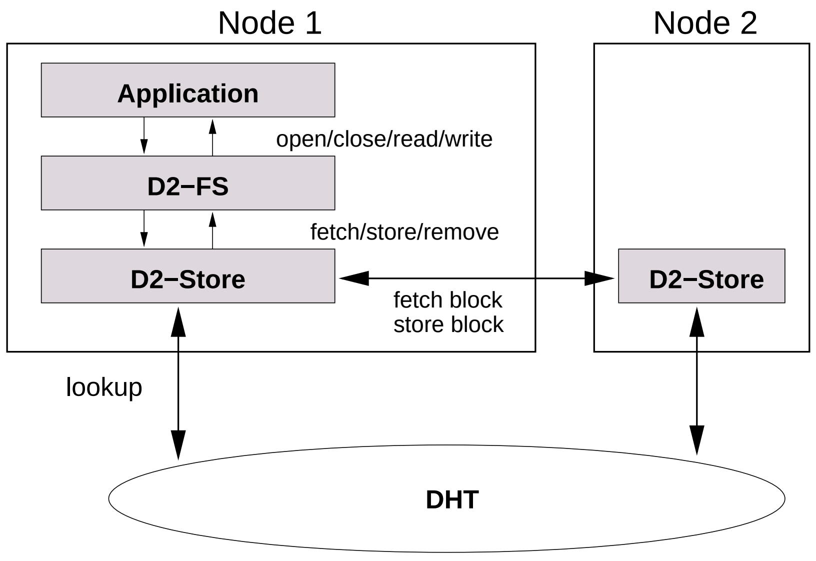


Figure 1: D2 architecture.

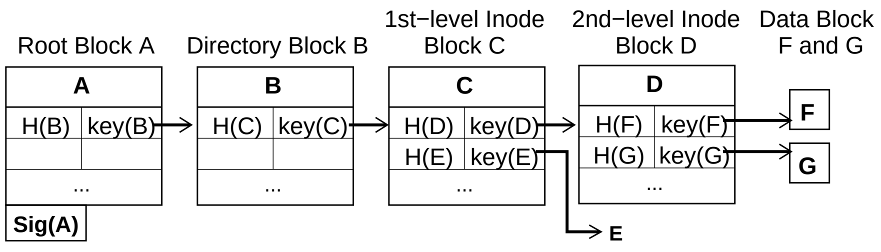


Figure 2: Data layout in D2-FS.

CFS [8]. D2 exports an NFS-like interface to users. Each volume in D2 is assumed to have a single writer at any given time, but may have multiple readers. Hence, D2 could be used for file system volumes containing shared files (e.g., binaries and libraries), user home directories, email storage, or large scientific datasets.

Major Components of D2. Each node in D2 consists of three primary components (Figure 1): a file system layer (D2-FS), a data storage layer (D2-Store), and a dynamic load balancing DHT component (DHT). D2-FS translates file system operations into block reads and writes, which are processed by D2-Store. D2-Store is responsible for data creation, retrieval, deletion, redundancy maintenance, and migration during load balancing. The dynamic load balancing DHT is used by D2-Store for block lookups. This DHT has the same scalable lookup properties as traditional DHTs, but it enables load balance of the non-uniform key distribution that is necessary for maintaining data locality.

D2-FS. To isolate the effects of defragmentation over other aspects of DHT-based file system design, D2-FS uses an organization of files similar to CFS instead of a completely new one. We give an overview of this organization here, noting aspects that are different due to defragmentation.

D2-FS maintains four types of blocks (Figure 2): a root block, metadata blocks for directories, file inodes, and file data blocks. All blocks are immutable, except the root block, which is updated in place. Furthermore, all of the metadata blocks are collectively signed by the publisher's public key. That is, the publisher signs the file volume's root block which contains pointers to the directory blocks; the directory blocks contain pointers to subdirectory blocks and inode blocks, and so forth. For each pointer, the block also stores a hash of the pointed-to block; therefore, signing the root block effectively signs all of the metadata. All blocks are at most 8 KB in size.

Each block is identified by a distinct DHT key. Each metadata block contains the DHT keys pointing to other metadata blocks or data blocks as in a standard file system. Since the relative closeness of keys in the key space determines the closeness of blocks placed in the system, key selection is central to preserving locality in D2. Section 4 describes this process in detail. In CFS, the key of a block is also its content hash, which also enables clients to verify the block's integrity. Keys in D2-FS are no longer content hashes in order to preserve locality, so each metadata block in D2-FS also keeps the content hashes of the blocks it points to, in order to enable integrity verification. To minimize the number of blocks used for each file, when the amount of file data in a data block is small enough, D2-FS stores the data directly in the parent metadata block.

When a file is updated, D2-FS inserts the new data blocks and then inserts new versions of all the metadata blocks along the full path to the root. This ensures that all readers will see an internally consistent view of the volume. This implies that each write must update all meta-blocks along the entire path. D2-FS avoids this overhead when writing temporary files by maintaining a 30-second write-back cache. The same cache is also used as a buffer cache so multiple reads of the same block occurring within a 30 second window only require the block to be fetched once. Due to this cache, data seen by users may be stale by up to 30 seconds, but a user will never see partial or incomplete writes.

D2-Store. D2-Store uses 8KB blocks as storage units to ensure reasonable storage load balance. Treating each file as a storage unit would improve locality for blocks within a file, but would not maintain locality across related files. Since D2-FS already ensures data locality, there is no need to treat files as units.

Each block is replicated on the r immediate successors of its key. The first is the primary replica and the remainder are secondary replicas. Erasure coding (with r fragments) could be used instead of whole block replication to save storage space at the cost of read/write performance and complexity. However, whether we use replication or erasure coding, defragmenting k objects so that they reside on r nodes instead $k \cdot r$ nodes achieves a similar availability improvement. Since we are interested in studying the effects of defragmentation and not replication techniques, D2-Store uses replication for simplicity. The qualitative comparisons we make between D2 and traditional DHTs should hold in either case.

D2-Store supports the same `put(key, data)` and `get(key)` interface as a traditional DHT, but also allows block removal (`remove(key, delay)`). Quick removal of deleted blocks is important for preserving data locality; if we leave unnecessary data in the system, it can fragment active data over more nodes than necessary by taking up space between active blocks. Since views can be stale by up to 30 seconds, D2-FS delays removal of blocks by 30 seconds. In addition,

Table 1: Workloads analyzed.

Workload	Duration	Accesses	Active Data	Description
HP (1999) [16]	1 week	238M	40GB	A block-level trace from a multi-disk research server.
Harvard (2003) [12]	1 week	60M	83GB	Research and email NFS trace. (EECS workload from [9].)
Web (2003) [17]	1 week	47M	93GB	Accesses to web sites seen by NLANR Web caches.

removal may fail in the rare cases when nodes are partitioned from the DHT, so blocks are also automatically removed after a user-defined TTL that can be refreshed.

4 Preserving Data Locality

Preserving locality in the assignment of data to nodes reduces the number of nodes a user must access to operate on their data. This section describes D2's approach for preserving data locality.

4.1 Requirements of Real Workloads

Although data locality obviously implies that blocks of the same file should be stored together, it is less clear how files should be clustered. The ideal clustering of the files depends on the groups of files are that are accessed together in tasks. To handle completely arbitrary tasks, a clustering algorithm would need foreknowledge of future access patterns. Even if the algorithm could learn from history, it would require access pattern prediction and dynamic file re-clustering.

To determine if there is a less complex design that would maintain most data locality in real file systems, we analyze three real workloads (Table 1). `Harvard` is the main workload we study. It contains timestamped accesses to an NFS server used by a large Harvard research group. `HP` contains timestamped accesses to blocks at the disk level. Each block has a number that corresponds to its position on the physical disk, though we do not know which blocks belong to the same file. Finally, `Web` contains timestamped Web accesses. Analyzing `Web` enables us to understand the data locality of a less traditional file system.

To estimate the data locality preserved by several designs, we analyze the number of nodes each user (or application, in `HP`) needs to access each hour, on average, for that design.[1] We consider three scenarios: **traditional**, **lower-bound**, and **ordered**. Each scenario assigns 250MB of data to each node. The **traditional** scenario corresponds to a traditional DHT-based file system that assigns uniformly random keys to data blocks. **lower-bound** is a lower bound on the number of nodes that a user needs to access, computed as the ratio of the total number of blocks accessed per user and the number of blocks stored on each node. This lower bound may not actually be achievable since a block placement corresponding to the lower bound could require a block to be on two different nodes (e.g., if two users access intersecting but non-identical sets of blocks that can not fit on a single node). In **ordered**, we assign keys that are consistent with the alphabetical ordering

[1]`Harvard` identifies users by `uid`; `HP` identifies applications by `pid`; `Web` identifies users by anonymized IP addresses.

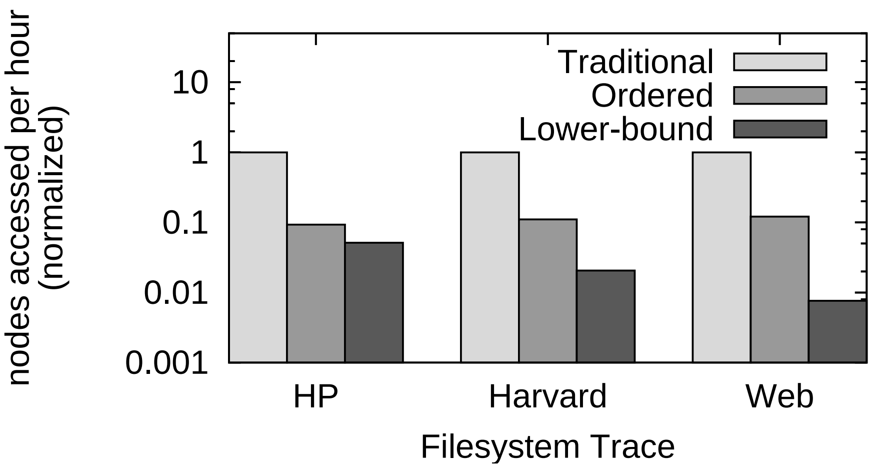


Figure 3: Mean nodes accessed per user each hour, normalized against Traditional.

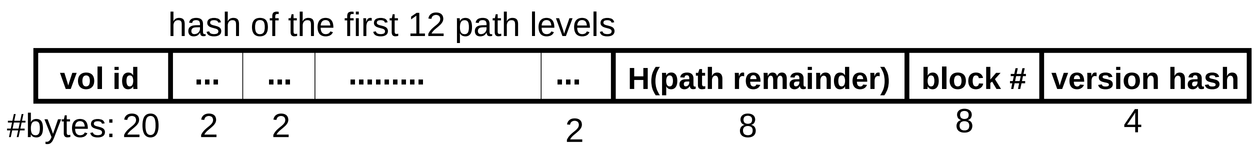


Figure 4: Key encoding for blocks in D2-FS.

of block names. For `Harvard`, the name of each block is the file's full path (including file name) and block number within the file. As a result, blocks of files in the same directory will have contiguous keys. In `HP`, the name of a block is simply its disk block number. Since local file systems tend to place blocks created at the same time near each other in the file system, blocks with close block numbers are more likely to belong to the same file or to files in the same directory. Finally, for `Web`, the name of each Web object is its URL with the domain name tuples reversed (e.g., the name for `www.yahoo.com/index.html` is `com.yahoo.www/index.html`). For this initial analysis, we make the simplifying assumption that each node stores the same number of blocks. Section 8 and 9 demonstrate D2's effectiveness despite the small load imbalance and temporary fragmentation that occurs when using D2's actual load balancing algorithm.

Figure 3 shows the results from this analysis. There are nearly two orders of magnitude difference between **traditional** and **lower-bound**, so there is significant potential for improving data locality. Compared to **traditional**, **ordered** reduces the number of nodes contacted by about 10 times. The difference between **ordered** and **lower-bound**, on the other hand, is less than an order of magnitude in both file system traces (though it is slightly larger in `Web`). Since the lower bound may not actually be achievable, we believe that the locality achieved by **ordered** is sufficiently close to the maximum possible to warrant further investigation.

4.2 Practical Locality Preserving Keys

The preceding analysis indicates that assigning keys that are consistent with the ordering of full path names (i.e., name-space locality) will likely achieve near-optimal data locality in file systems. We could use path names directly as keys. However, DHTs usually use fixed sized keys, so every lookup message would have to contain a key that is as large as the longest path. Thus, we use a more compact key encoding, shown in Figure 4, to limit message overhead without modifying DHT routing and maintenance logic.

The first 20 bytes encode the ID of the volume that a block belongs to. The next 24 bytes encode the file's path, with each directory in the path encoded with 2 bytes. When a file is added to a directory, an unused 2-byte value in that directory is assigned to the file; an unused value is found by examining the existing file list in the directory block.[2] Since 24 bytes is only enough space for 12 directories, for longer paths, the next 8 bytes are a hash of the remainder of path. Although locality for files in such long paths will not be preserved, they make up less than 1% of the files in both the `Harvard` and `Web` workloads and an even smaller percentage of the accesses. The next 8 bytes are allocated for file inode and data block numbers. Finally, the last 4 bytes are used to distinguish different versions of an overwritten block, so that slightly stale views can still access the old versions, as in CFS. We choose this key representation for simplicity; namespace flattening schemes [14] can be used to make it more compact.

We believe that this encoding offers a good trade off between key size and representativeness. This 64 byte key enables 64K files per directory and files that are up to many exabytes in size and is more than sufficient for the workloads we analyzed.

This key encoding enables naming of new files and directories, but a file in D2-FS may also be moved to a different directory. If the keys of this file's blocks are also changed to reflect the new path, the blocks will need to be moved to the new corresponding locations in the DHT, causing significant churn in the key distribution if the moved file is large or is a directory containing a lot of data. Instead, D2-FS keeps the original keys for renamed files; the file's new parent directory simply points to the file's original location. Excepting file moves immediately after creation, which are stored in D2-Store only after the move due to D2-FS's writeback cache, file moves are rare (only 0.05% of the operations in the `Harvard` workload), so their impact on data locality should be minimal.

4.3 Discussion

Although we show that preserving data locality improves overall performance and availability, it also raises three concerns in extreme scenarios. First, whereas the failure of a replica group (r consecutive nodes) in a traditional DHT results in the unavailability of *some* data for *many* users, the same failure results in the unavailability of *most* of the data for a *few* users in D2. This trade off is unavoidable, but our evaluation shows that for a typical file system workload, isolating failures to impact a smaller number of users (at the cost of affecting more data per user) results in a much lower user-perceived failure rate for *all* users. Second, when all of the nodes in the replica groups

[2] An application that encodes a file's path without knowledge of its parent directory, such as a Web cache, can use a 2-byte hash of each directory name instead, losing a small amount of locality when there are collisions.

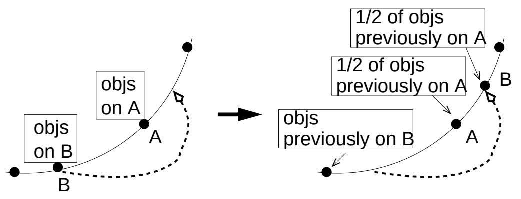


Figure 5: Load balancing example.

that hold a particular user's data are far away in the network relative to the user, that user will experience high latency for all his accesses. However, our evaluation shows that this scenario is rare and has minimal impact on relative performance compared with a traditional DHT. Finally, blocks with identical content but addressed using different keys will be stored on multiple nodes in D2, whereas they would only be stored once if they were addressed using content hashes, since their hashes would be the same. This increased storage cost is a necessary for preserving data locality because the existence of multiple keys for a block indicates that the block can be accessed as part of multiple segments of data; this cost is probably acceptable in most circumstances given the low utilization and cost of disks today.

5 Caching DHT Lookups

D2 maintains locality in data placement, so each user should not usually require data from many nodes. This observation alone, however, does not reduce the number of DHT lookups users must perform. D2-Store avoids lookups using a *lookup cache*. The cache stores the IP addresses and key ranges of nodes in recent lookup results. Future requests that access keys in cached key ranges circumvent the lookup step entirely. Clients could also use a lookup cache in a traditional DHT, but it would be much less effective since future requests are not as likely to access keys in recently accessed key ranges.

When nodes join and leave the system, cache entries can become stale. Since D2-Store will fall back to a normal lookup when a block is not found, using a stale cache entry does not affect correctness, but it does hurt retrieval latency. Therefore, D2-Store evicts cache entries after 1.25 hours, based on the leave/join rate of PlanetLab nodes during the week used in our evaluation [5] (Section 8.1). In other environments, the cache entry TTL could be changed dynamically based on measured cache invalidation rates.

6 Load Balancing

Load Balancing Algorithm. Since D2's key distribution is no longer uniform, consistent hashing cannot be used to preserve load balance. D2 instead uses a dynamic load balancing DHT [4, 21]. These DHTs were originally designed to support range queries, but D2 uses their load balancing

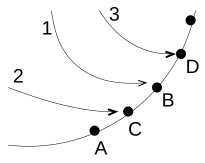


Figure 6: Example of unnecessary data transfers during load balancing (see text).

mechanisms specifically to preserve locality in data placement. These load balancing algorithms are simple, fully distributed, and converge quickly.

We present a brief description of the basic algorithm here (refer to Figure 5). Each node B periodically contacts another random node A in the system (once every *probe interval*). If the load on A is greater than t times the load on B, B changes its ID to become the predecessor of A, effectively taking half of A's load. The ID change is implemented by having B leave the DHT and then rejoin with the new ID. We leave some details to [21]. For $t \geq 4$, all the nodes achieve a load that is a constant multiplicative factor away from the average in $O(\log n)$ steps with high probability [21]; we use $t = 4$ so that node loads differ by at most a factor of 4 in steady state. Our D2 prototype uses the Mercury DHT [4], which implements a version of this algorithm using an efficient random sampling technique and maintains $O(\log n)$-hop routes in the DHT.

Each node stores both primary and secondary replicas, but only the primary replica count is used as the load value for the purpose of this algorithm because ID changes only directly affect the primary replica count. To maintain the invariant that the a block will be stored on the r nodes succeeding its key after a load balancing operation that moves node B, the r nodes succeeding B's old and new positions fetch required replicas that are not already present and delete unnecessary replicas. When primary load on all nodes is balanced, then total load, including both primary and secondary replicas, will be balanced as well.[3]

D2 uses Mercury to balance storage load, but request load (i.e., the number of block downloads serviced by a node) is also important because some files may be accessed more than others. D2's use of Mercury to balance storage load is orthogonal to traditional caching techniques to balance request load, so D2 alleviates temporary hot spots using retrieval caches like traditional DHTs [38], thereby balancing both storage and request load.

Reducing Load Balancing Overhead. During load balancing, a block can be moved multiple times. Suppose that node A is heavily loaded, and node B changes its own ID to become A's predecessor, taking half of A's load (Figure 6, 1). A may still be heavily loaded (and now B as well), so later C (Figure 6, 2) and D (Figure 6, 3) change their IDs to become the predecessors of A and B, respectively. Now B must transfer half of its blocks to D; these blocks originated from A. Thus, these blocks move twice. This often occurs when a large file is inserted, since it will initially be assigned to a single node.

To minimize duplicate data movement D2-Store uses *block pointers* during load balancing. Instead of having A immediately transfer half of its blocks to B when B becomes A's predecessor,

[3]Suppose the ratio of maximum primary load max to minimum primary load min in the system is c. Then the ratio of maximum total load to minimum total load is also at most $\frac{r \cdot max}{r \cdot min} = c$.

B will initially maintain block pointers to A. Later B will transfer the pointers to D, and D will ultimately retrieve the actual blocks from A and delete the pointers. A node will retrieve the block for a pointer when it has held the pointer for longer than the *pointer stabilization time*.

Using block pointers only temporarily hurts data locality during load balancing; our evaluation shows their impact on availability is small. In addition to reducing load balancing overhead, pointers also enable writes to succeed even when the target node is at capacity; as in PAST [38], pointers can be used to divert blocks from full nodes to those with space (assuming a small amount of space is always left over for pointers). However, in D2, the node at capacity will eventually shed some load when it performs load balancing, so this additional indirection is temporary.

7 Prototype Implementation

D2 is implemented in C++ and uses libasync [27] for asynchronous event processing. D2-Store stores data blocks in BerkeleyDB [2] and uses Mercury [30] for load balancing and DHT lookups. D2-Store uses TCP for communication while Mercury uses UDP. Mercury performs recursive DHT lookups but D2-Store downloads blocks directly from the responsible nodes once they are located. Our implementation supports all the aforementioned client operations, though it does not yet perform authentication for writes and removals.he implementation consists of about 13K lines of code.

Our prototype of D2-FS implements the core functions described in this paper (such as key encoding, file system layout, and write-back cache), but does not yet have an interface to higher-level applications. The lack of an interface does not affect our results because our evaluation is based on the replay of existing file system traces and would bypass this interface anyway. The traditional DHT we compare D2 against in the evaluation uses the same code base as D2-Store, D2-FS, and Mercury, but uses hashed keys and consistent hashing instead of locality preserving keys and dynamic load balancing.

8 Availability

As discussed in Section 1, user and application tasks may access multiple objects and will fail if any one of these objects is unavailable. We evaluate the availability (i.e., success rate) of tasks instead of the availability of individual objects because the former metric more accurately reflects how often users will perceive the system to have failed to complete a unit of work.

8.1 Experimental Setup

Testbed. In order to make evaluation of task availability over long periods of time tractable, we developed a detailed event-driven simulator. The simulator models a 750kbps per-node bandwidth limit on load balancing traffic (i.e., data migration). Network latency is ignored since RTTs are orders of magnitude smaller than the time scale of events that affect availability (e.g., MTTF, MTTR, and data-transfer time). Each user writes data into the system at 1500kbps. The load

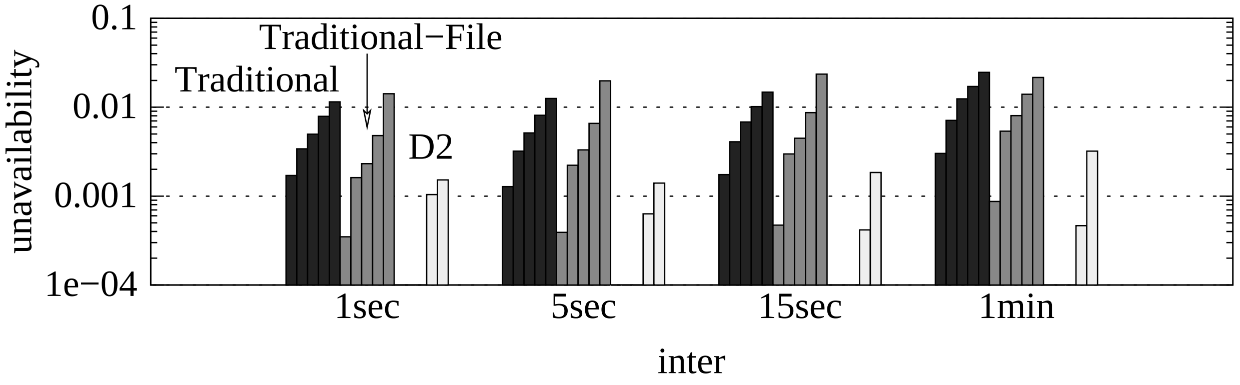


Figure 7: Task unavailability under each system while varying $inter$.

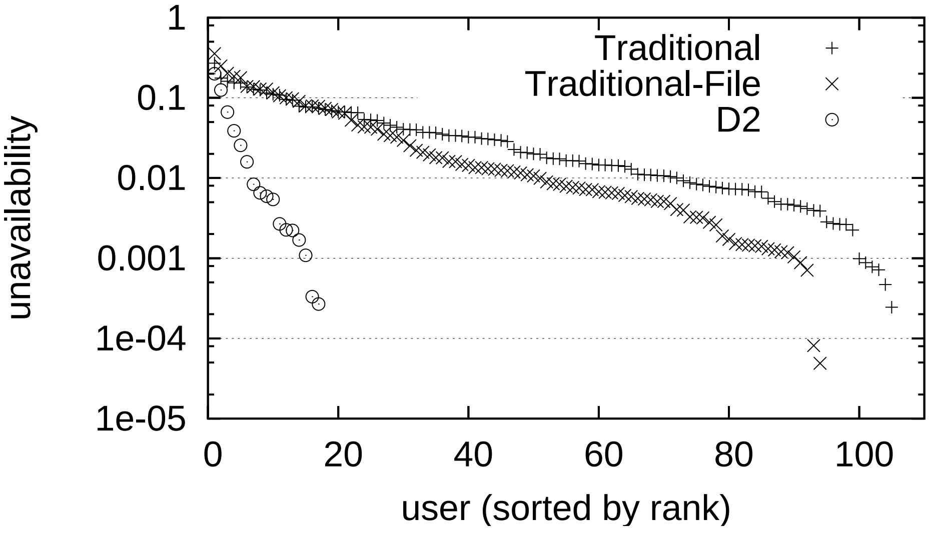


Figure 8: Unavailability experienced by individual users, ranked by decreasing unavailability, when $inter$ = 5sec. Users not shown do not experience any unavailability.

balancing probe interval is 10 minutes and the pointer stabilization time is 1 hour. Each object has 3 replicas.

The simulator captures all facets of D2 except DHT routing. Although a routing failure can cause a request to fail even when the replicas that satisfy the request are available, this failure is transient and can be resolved by retrying after a delay comparable to the DHT link repair time, which is usually short. Moreover, because the churn rate is low in the failure model we use (see below), the duration of all link inconsistencies would be orders of magnitude smaller than node down times.[4] Hence, the rate of link inconsistencies caused by churn in D2 is comparable to a traditional DHT.herefore, replica availability is the primary factor for task availability.

Workload and Failure Model. Our experiments simulate 247 nodes based on the observed failure behaviors of the same number of nodes on PlanetLab [5] from February 22 to 28, 2003, a week with a particularly large number of failures (see [35] for details). By using an empirical trace with large failure events, failure correlation is captured more realistically, which is the most likely factor to reduce availability in practice due to its unpredictability.

We evaluate D2 using the `Harvard` workload (see Table 1) because it is the only trace that has file path information and file writes (i.e., modifications, creations, and deletions). The former is needed to use D2-FS's key encoding, and without the later, dynamic load balancing would not be needed. Each simulation is initialized by inserting all files that exist at the beginning of the trace into the DHT. The load balancing process is then simulated for 3 days so that node positions stabilize with respect to the initial key distribution.

Tasks. The `Harvard` trace does not contain any information that would allow us to definitively correlate the accesses in each task. Thus, we approximate a task as a sequence of accesses by the same user where the time between any two consecutive accesses is less than an inter-arrival threshold $inter$. We limit task duration to 5 minutes. Tasks defined with short $inter$ values represent application operations that do not include human interaction, while tasks defined with longer $inter$ values represent several consecutive user actions that may be correlated.

8.2 Task Availability

Figure 7 plots the task unavailability (i.e., the fraction of tasks that fail) of D2 versus that of a traditional and a traditional-file DHT. Each bar corresponds to one of 5 trials (each initialized with random node IDs). The 3 missing bars for D2 indicate trials with no unavailability. D2 decreases unavailability by an order of magnitude for all $inter$ values, in terms of the average, maximum, and minimum of the 5 trials. The traditional-file DHT achieves lower average unavailability than the traditional DHT by storing blocks of the same file on a single node, but it does not preserve as much locality as D2 for tasks that access many files. Figure 8, which plots the task unavailability experienced by individual users in the 5sec,5min case, shows that task failures also affect a smaller number of users. Increasing the number of replicas benefits D2 more; for example, with 4 replicas, D2 had no failures in all 5 trials while the traditional system had at least 3×10^{-6} of its tasks fail in 4 trials.

For insight into this result, consider the average number of nodes accessed per task in a traditional DHT. Table 2 shows the mean number of blocks and files required by a task and the mean number of nodes accessed by a task in each system. Suppose p is the probability that at least one node in a replica group (3 consecutive nodes in the DHT) is available. A traditional DHT requires the availability of about 10 to 23 replica groups per task, so the success rate is approximately p^{10} to

[4]Node leaves and joins caused by load balancing are voluntary, so DHT routing geometry can be repaired immediately.

Table 2: The mean number of blocks and files accessed per task, and the mean number of nodes accessed per task in the traditional DHT (block), traditional-file DHT (file), and D2.

	mean objects		**mean nodes**		
inter	**block**	**file**	**block**	**file**	**D2**
1sec	63	10	10	6	2
5sec	91	15	11	8	2
15sec	128	22	14	10	3
1min	237	38	23	16	4

p^{23}.[5] In contrast, the average task accesses only 2–4 replica groups in D2, so the task success rate is at least p^4. Hence, the expected failure rate of the former ($1 - p^{10}$ to $1 - p^{23}$) is much larger than the failure rate of the later ($1 - p^2$ to $1 - p^4$). In the PlanetLab failure trace, the probability that all nodes in a replica group fail is 0.02 without replica regeneration, so $p \geq 0.98$ in our experiments since regeneration increases the likelihood that a replica group will be available.ince tasks access multiple objects, whether data is stored as block or file units in a traditional DHT, the availability of more nodes are likely to be required to complete a task when compared to D2.

9 Performance

D2's defragmented design affects two primary aspects of system performance: DHT lookup traffic and user-perceived end-to-end latency. This section evaluates how defragmentation affects lookup traffic and end-to-end latency with our implementation and a real file system workload.

9.1 Experimental Setup

Testbed. We evaluated our implementation of D2 on 50 "PC 3000" machines in the Emulab testbed [39] running Linux 2.4.31, with each physical machine hosting multiple instances, effectively yielding systems of 200 to 1,000 virtual nodes. Kernel-level packet queues (based on a modified Linux version of WaspNet [33]) are used to emulate the wide-area network topology connecting these nodes. The topology accurately models pairwise end-to-end latencies between all virtual nodes, which we base on measured latencies between several thousand local DNS servers around the world [25]. In addition, the topology models per-node access link capacity limitations of 1500kbps or 384kbps. The first capacity captures service limits on infrastructures such as PlanetLab [5], while the second models scenarios where nodes are more constrained. Since both these speeds are much smaller than speeds in the Internet's core and our workload does not exhibit much contention, we do not model cross traffic. Finally, user-perceived performance will only be affected when the DHT is the bottleneck, so we do not limit the download bandwidth of clients.

[5]The actual probability is slightly higher since every r replica groups overlap, but the number of groups accessed is much less than n/r so this difference is negligible.

We compare D2 with a traditional and a traditional-file DHT implemented with the same codebase but using consistent hashing for block/file assignment. We allow the traditional-file DHT to do partial reads and writes on files, so all three systems read and write the same volume of data. For a fair comparison, the file block organization in all systems is identical to that described in Section 3; the traditional-file DHT simply stores all the blocks in a file on a single node.ll systems have 4 replicas per object.

Workload. For the reasons discussed in Section 8, we use the `Harvard` workload (see Table 1) in evaluating performance. Moreover, because D2-FS uses a write-back cache, we only evaluate end-to-end read performance (as was done in [7, 8]).

To make our experiments tractable yet representative of the entire trace, we evaluate system performance during 8 different 15-minute periods, chosen randomly from the five 9 AM to 6 PM periods. For each time period, we first simulate the DHT ring geometry, DHT node contents, and each user's range-based lookup cache content from the beginning of the workload to the start of the time period, using the simulator described in Section 8. Then, we initialize the Emulab deployment based on the simulation output before injecting the 15 minute workload. Each of the 83 users active during these times is assigned to a randomly chosen node. In addition, to emulate a more optimized DHT transport protocol such as STP [7], we establish TCP connections between each pair of nodes before replaying each workload segment. This benefits the traditional DHT more because it requires downloads from more nodes and, thus, avoids TCP connection setup more often. Hence, our comparisons between D2 and our traditional DHT are pessimistic.

In experiments with 200 nodes, we use the `Harvard` file system directly, with 4 replicas for each data block. In experiments with larger system sizes, we scale up the workload accordingly by replicating the initial file system. That is, we have 5.5 million blocks in the 200 node experiment, so in the 1000 node experiment, we add four extra copies of the file system, yielding 27.5 million total blocks. Since we only have 83 distinct access patterns, we still only replay accesses from 83 users.

Access Groups. One limitation of the `Harvard` trace is that it does not contain inter-access dependencies, which directly impact the amount of exploitable parallelism. A request R_2 that is dependent on an request R_1, cannot be submitted until R_1 completes. For example, a file cannot be fetched before the directory that contains it, since we will not know its key. However, if R_1 and R_2 are not dependent, such as when a read covers two consecutive blocks of the same file, they can both be submitted in parallel.

Instead of trying to recover inter-access dependencies, our evaluation considers two extremes. For a given user, we consider any period between two consecutive accesses that is larger than 1 second to be *think time*. Accesses with think times between them cannot be parallelized, so we leave think times unchanged. We define an *access group* to be the set of accesses that fall between two consecutive think times. At one extreme, denoted by **seq**, we assume that all accesses in a group are dependent and hence must be issued sequentially (i.e., one must complete before the next is submitted). At the other extreme, denoted by **para**, we assume that no accesses in a group are dependent and, hence, all can be issued in parallel. The actual amount of exploitable parallelism will be between these extremes.

In practice, we found that a large number of active parallel lookups and TCP downloads ini-

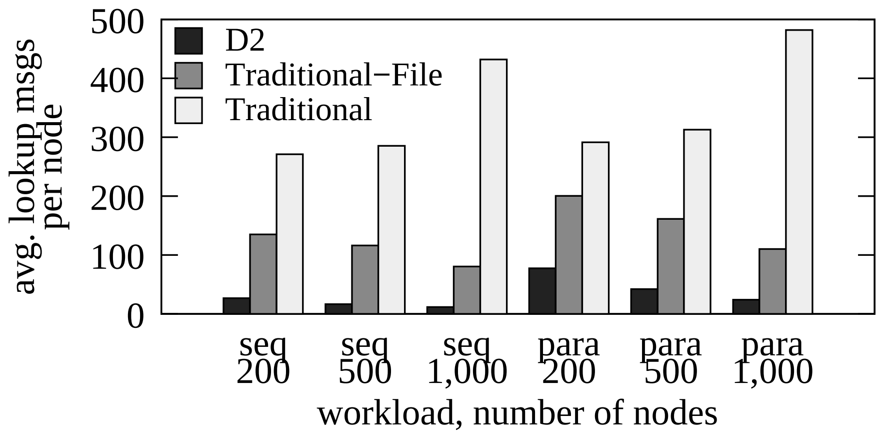


Figure 9: Reduction on DHT lookup traffic in D2.

tiated by the same node interfered with each other enough to incur 10 second application-level timeouts, primarily due to the low bandwidth of DHT nodes in our evaluation scenarios. Based on empirical measurements, we limit the number of simultaneous transfers each client can initiate to 15 in order to minimize the impact of these timeouts on performance. Although this limitation restricts the maximum number of nodes a client can download from, the number is still 3.75 times more than can be exploited from a single replica group, which has only 4 nodes.

9.2 Lookup Traffic

Figure 9(a) compares the average number of lookup messages sent by a node in D2 compared with a traditional system in DHTs of varying sizes. As expected, D2 substantially reduces the amount of DHT lookup traffic. With 1000 nodes, in both **seq** and **para** cases, D2 incurs less than $1/20$ of the lookup traffic in a traditional system.

To gain further insight, consider the following estimate of the average number of lookup messages per node:

$$\frac{\text{\# accesses} \times \text{ lookup cache miss rate} \times \text{ msgs per lookup}}{\text{\# nodes in system}}$$

Figure 9 shows that lookup traffic per node in a traditional DHT *increases* with system size, despite the number of accesses remaining the same. This increase occurs because the total cache miss rate increases dramatically with system size (for example, by 66% when scaling from 200 to 500 nodes). However, lookup traffic per node *decreases* with system size in D2. This effect is explained by two factors. First, the number of messages per lookup only grows logarithmically with system size. Second, the cache miss rate is mostly independent of system size, since the data a user accesses will likely be on the same number of nodes in all systems due to locality. Thus, the numerator in the ratio above grows more slowly than the denominator, so D2 is more effective at using additional resources to reduce lookup overhead than a traditional system.

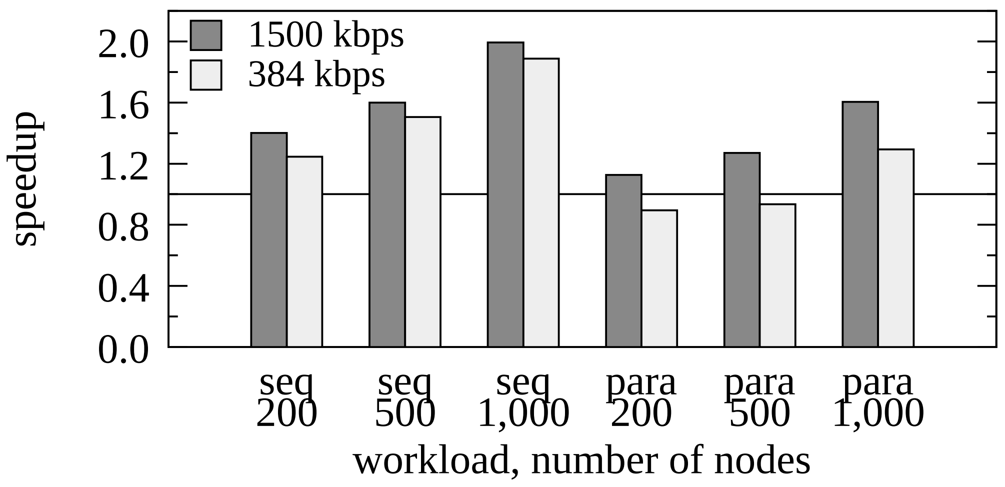


Figure 10: Speedup over a traditional DHT.

Interestingly, the number of lookup messages in a traditional-file DHT also decreases with system size, though the number is still much larger than that seen in D2. This effect is also due to the fact that the cache miss rate in a traditional-file DHT does not grow appreciably with system size. The cache miss rate is more stable in a traditional-file DHT than a traditional DHT because a user's working set of files is much smaller than a user's working set of blocks (see Table 2). A user will need to lookup at most as many nodes as the size of this working set.

Although lookup traffic is usually a small fraction of data block transfers, lookup services that store most data on application-specific nodes external to the DHT (such as OpenDHT [37]), would benefit greatly from the reduction in lookups. Moreover, even the small amount of bandwidth saved due to the reduction in lookup messages can be used effectively for additional route discovery and link maintenance [26], improving both end-to-end performance and DHT routing robustness.

9.3 End-to-end Performance

Speedup. Each access group represents a unit of work between user think times. Hence the completion time of access groups is the latency that users will perceive. We compare the end-to-end performance of D2 to a traditional DHT using the *speedup* of each access group, or the ratio between the access group's completion time under a traditional system and its completion time under D2. Speedup is greater than 1 when D2 is faster and less than 1 when it is slower. For each user, we compute the geometric mean speedup across all access groups of that user. The overall speedup of the system is the geometric mean speedup across all users.[6]

Sequential Performance. Figure 10 shows the average speedup of D2 over a traditional DHT with different system sizes and DHT node access bandwidths. As expected, in the **seq** case, D2 always achieves noticeable speedups over a traditional design. For example, in a system size of

[6]The average is computed using a geometric mean since we are averaging ratios.

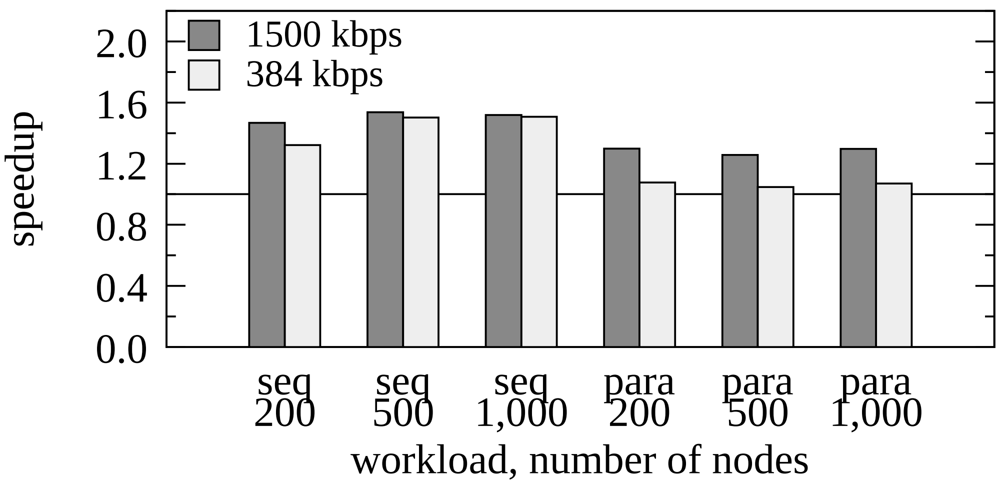


Figure 11: Speedup over a traditional-file DHT.

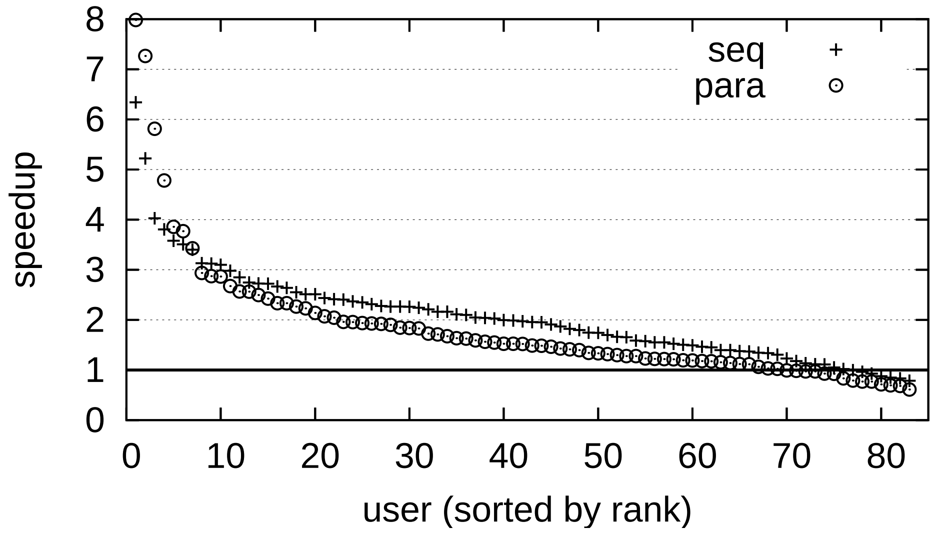


Figure 12: Mean speedup over a traditional DHT for each user in the 1000 node, 1500kbps scenario.

1000, the performance improvement is at least 90%. Figure 11 shows the average speedup of D2 over a traditional-file DHT. Since access groups often require multiple files (see Table 2), the sequential speed up is similar in a 200 node system. However, in contrast with the traditional DHT comparison, the speedup does not grow appreciably with system size (see the cache discussion below).

Figure 12 shows the breakdown of improvement over a traditional DHT for each user in the 1000 node, 1500kbps scenario. Nearly half of the users actually see an improvement larger than the

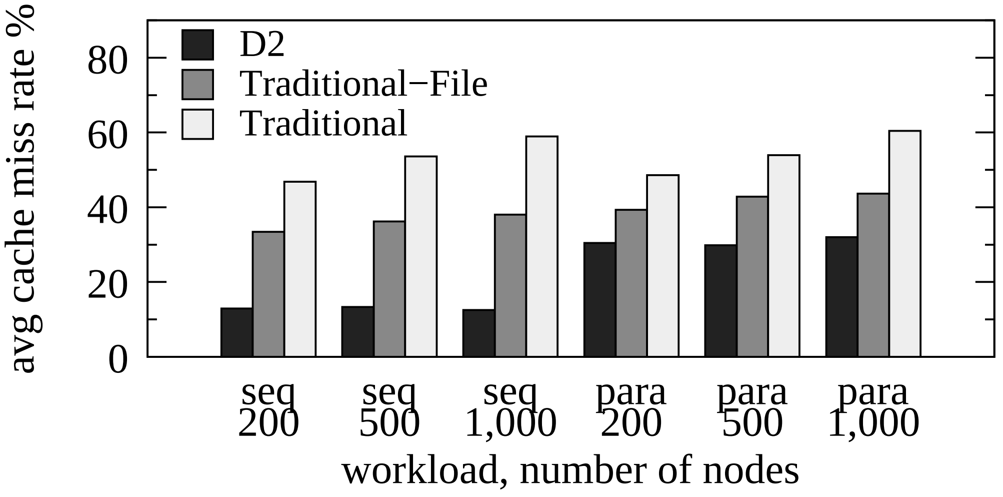
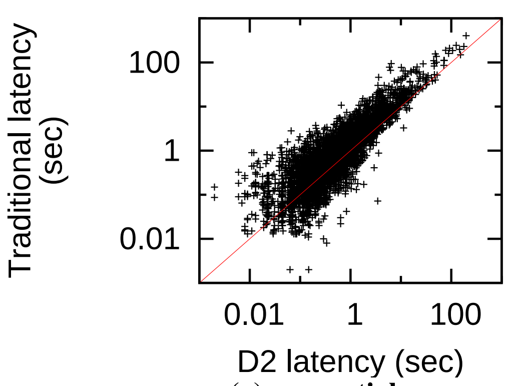


Figure 13: Mean lookup cache miss rate.

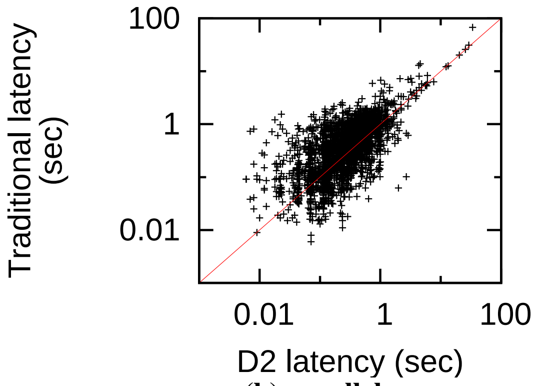
mean in the **seq** case. However, 6 users see a performance degradation. This degradation occurs when many replicas of a user's data are assigned to distant nodes in the network. Nonetheless, the slowdown is small relative to the speedup that most users see. In addition, this effect can be mitigated by always downloading blocks from the closest replica, since there is usually at least one that is not distant (D2 currently selects replicas randomly).

The lookup cache miss rate is primarily responsible for the speedups we observe. Figure 13 shows the average per-user lookup cache miss rates of each system in each of Figure 10's scenarios. In the **seq** cases, D2 has miss rates of 13% while the traditional design has miss rates of more than 47%. Moreover, as explained in Section 9.2, the miss rate for D2 is independent of system size, while the miss rate in a traditional DHT grows with system size. The traditional-file DHT maintains some locality by storing all a file's blocks on one node, so its cache miss rate also does not grow appreciably with system size. This is most likely because each user's working set of files is small (i.e., smaller than 200, the number of nodes in the smallest DHT system), so only the nodes hosting these files need be cached in a traditional-file DHT. However, there is likely a sufficient number of blocks in this working set to require accessing more nodes in a traditional DHT when it grows (i.e., more than 200 or 500 blocks).

Finally, Figure 14(a) compares the latency of access groups under D2 and the traditional DHT in the 1000 node, 1500kbps scenario. Figure 15 shows that the results comparing D2 with a traditional-file DHT are similar. Points above the diagonal complete faster in D2, while points below complete slower. The weight of the distribution is above the diagonal. Most points below are access groups that take between 0 and 2 seconds in both systems and, since inter-node latencies vary by several 100 milliseconds, can mostly be attributed to blocks being assigned to nodes closer in the network when using the traditional DHT. More importantly, most access groups that take more than 5 seconds to complete in either system complete faster in D2, sometimes by almost an

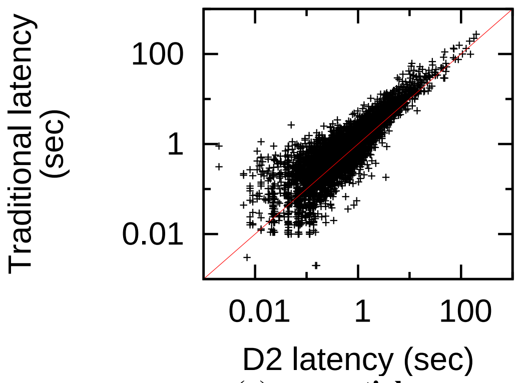


(a) sequential

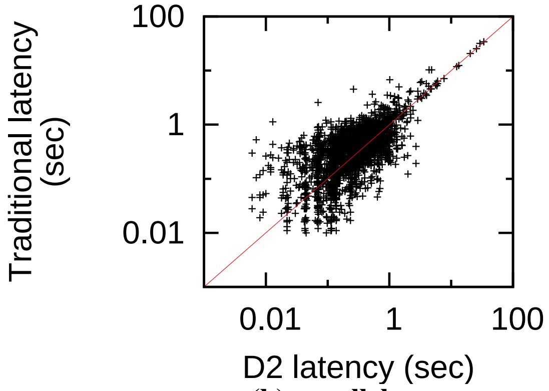


(b) parallel

Figure 14: Comparison of access group latencies for D2 and the traditional DHT. Note the logarithmic scales.



(a) sequential



(b) parallel

Figure 15: Comparison of access group latencies for D2 and the traditional-file DHT. Note the logarithmic scales.

order of magnitude.

Parallel Performance. Figure 10 shows that the average speedup in the **para** case is greater than 1 in all system sizes when nodes have access bandwidths of 1500kbps. Hence, D2 still out-performs the traditional DHT, though not by as large a margin as in the **seq** case. The per-user breakdown of speedup measurements (see Figure 12) in the **para** case is also similar to the **seq** case, which is again explained by the difference in cache miss rates (see Figure 13).However, when the access bandwidth is reduced to 384kbps, D2 performs worse than the traditional DHT when the system size is 200 or 500 nodes. In a traditional DHT, the requests in an access group will likely be serviced by more nodes than in D2, since blocks are assigned to nodes randomly. Thus, although each individual access may take longer due to lookup cache misses, different accesses in the same group can simultaneously leverage the upload bandwidth of more nodes in a traditional DHT.

When per-node access bandwidth is low, as in these two scenarios, the throughput gain outweighs the additional lookup latency. Nonetheless, when the system size increases to 1000 nodes, the lookup latency, which grows with system size, again dominates and the speedup is greater than 1.

Figure 11 shows that D2's parallel speedup over a traditional-file DHT is greater than that over a traditional DHT when there are 200 nodes in the system. This is because, unlike D2, access groups that require multiple related files still access multiple nodes in a traditional-file DHT, while access groups that read very large files can not take advantage as much parallelism as a traditional DHT because all the blocks of a file are stored on the same set of nodes. Since most access groups that require more than one block fall into one of these two categories, the traditional-file DHT's average parallel performance is the poorest when there are few nodes. However, unlike the traditional DHT, the traditional-file DHT's cache miss rate does not grow appreciably with system size (see Figure 13), so its parallel performance does not degrade with system size as with a traditional DHT. Nonetheless, D2 still out performs the traditional-file DHT consistently.

Figure 14(b) shows that there are many access groups that take longer to complete in D2 when compared to a traditional DHT. However, the weight of the distribution is still above the diagonal. Moreover, no access group that takes more than 5 seconds to complete in D2 complete much faster in the traditional DHT — they all lie close to the diagonal. Although we expect these access groups to perform much better in the traditional DHT because they contain many parallelizable requests, a subtle limitation of the TCP transport prevents the full upload bandwidth from each node from being utilized.

When a TCP connection is idle for more than one retransmit timeout (RTO) it reduces its window size and re-enters the slow start mode of operation. Consider a traditional DHT in the 1000 node, 1500kbps scenario. The first 15 requests in flight each require the downloading TCP connection to enter slow start; with 8KB blocks and 1500 byte packets, this means that at least 2 RTTs are required to fetch a block.[7] The mean RTT in our network is 90ms, so the average node effectively only transfers at a rate less than 309kbps. In addition, the expected time between accesses to the same node is at least 14 seconds,[8] much longer than one RTO. Thus, the average block download will *always* require the TCP connection to enter slow start and, hence, will never be able to utilize the DHT node's full bandwidth. In D2, since most requests are likely to go to the same 4 replica nodes due to locality, each TCP connection can usually ramp up to the full 1500kbps.

Custom DHT transfer protocols like STP [7] are designed to avoid this adverse interaction between large parallel downloads and TCP. STP improves by eliminating connection setup and using a single congestion window for all connections to avoid per-flow slow-start. However, the traditional DHT used in our experiment does not suffer from the lack of these two attributes in most cases and in some circumstances may actually perform better. In regards to connection setup,

[7]In Linux, the sender's window begins at 2 packets (2920 data bytes), so the sender cannot reply to a request with the entire 8KB block at once. It must wait one RTT for the first 2920 bytes to be acked before sending the rest. In addition, there is potential lookup and transfer time.

[8]We can estimate the number of 15-request batches until a node is accessed again as a geometric distribution with $p = \frac{15}{1000}$. Thus the expected number of batches between accesses is $\frac{1}{p} = 66.7$ and each batch takes at least 2 RTTs plus transfer time, or 212ms, to complete [34]. More than 212 requests must be in flight at once to reduce the expected time between accesses to the same node to less than 1 second.

recall that in our traditional DHT, a TCP connection is pre-established between each pair of nodes to emulate such an optimized protocol, so there is no setup for downloads in our traditional DHT either. Although downloads in our traditional DHT do perform per-flow slow-start, since each node can fetch 15 blocks at the same time and 86% of access groups in our experiment access at most 15 blocks, most access groups would actually complete faster than when using a single congestion window. Moreover, when using a single congestion window, throughput from all destination would suffer if the path to any one contains a bottleneck [1], and TCP has more fine-tuned loss recovery mechanisms than STP. Therefore, using proposed custom transfer protocols would not substantially improve the traditional DHT's overall parallel download performance in the scenario we examine. Furthermore, optimizing the transfer protocol would not improve a traditional DHT's availability, lookup message volume, or sequential download performance. D2 improves all these characteristics while maintaining good parallel download performance in the common case.

10 Load Balance and Overhead

The primary cost of D2's availability and performance gains is the need for active load balancing. As files are added, removed, and modified, the key distribution of blocks in the system changes, and D2 may have to migrate blocks in order to maintain load balance. This requirement poses two questions: (1) Can D2 maintain storage balance over time? (2) How much network traffic is required to maintain this balance?

Workloads. We seek answers to these questions by performing long term simulations using a setup like that described in Section 8.1 with the `Harvard` workload. In order to isolate data movement overhead caused by load balancing from that caused by replica regeneration, we do not simulate node failures. However, we note that introducing the failure model described in Section 8.1 did not appreciably change our results.

In order to study the load balance and overhead properties of D2 under extreme conditions, we also evaluate a workload that uses the DHT as a Web cache, like Squirrel [18]. When a client requests a URL, it first attempts to fetch the file from the DHT; if not available, the data is downloaded from the actual Web server to the client and inserted into the DHT, so that the next client that requests the URL can fetch the cached copy. With a traditional DHT, the file's key is a hash of the URL. With D2, the file key is the URL encoded using the scheme described in Section 4.2. This Web cache workload, which we denote as `Webcache`, is simulated using real Web cache traces collected between May 20 to 26, 2005.[9]

We stress that we do *not* believe that a Web cache would benefit substantially from using D2 over a traditional DHT, since higher unavailability merely results in a higher cache miss rate, and the target deployment scenario is usually a high speed LAN where download performance cannot be improved substantially. We only evaluate this application to illustrate that D2 can handle a workload with an extremely high data churn rate. Load balance is much harder and more costly to maintain when file insertions and evictions dominate the workload (as in this one) since the data

[9]We use traces from `rtp`, the NLANR cache with the largest traffic volume. See `Web` in Table 1 for details. Only cachable content is cached in the DHT, which is detected with techniques from [18], and the DHT evicts cached content that is not refreshed in one day or is replaced with a newer version fetched by a client.

Day	**1**	**2**	**3**	**4**	**5**	**6**
`Harvard` W_i/T_i	0.10	0.20	0.16	0.16	0.17	0.18
`Webcache` W_i/T_i	0.09	1.29	13.3	0.98	0.87	0.79
`Harvard` R_i/T_i	0.10	0.22	0.16	0.13	0.10	0.11
`Webcache` R_i/T_i	1.01	1.15	1.31	1.02	1.03	1.02

Table 3: The ratio of data written W_i and the total present at the start the day T_i on each day i, in terms of byte volume. Also the ratio of data removed R_i and T_i.

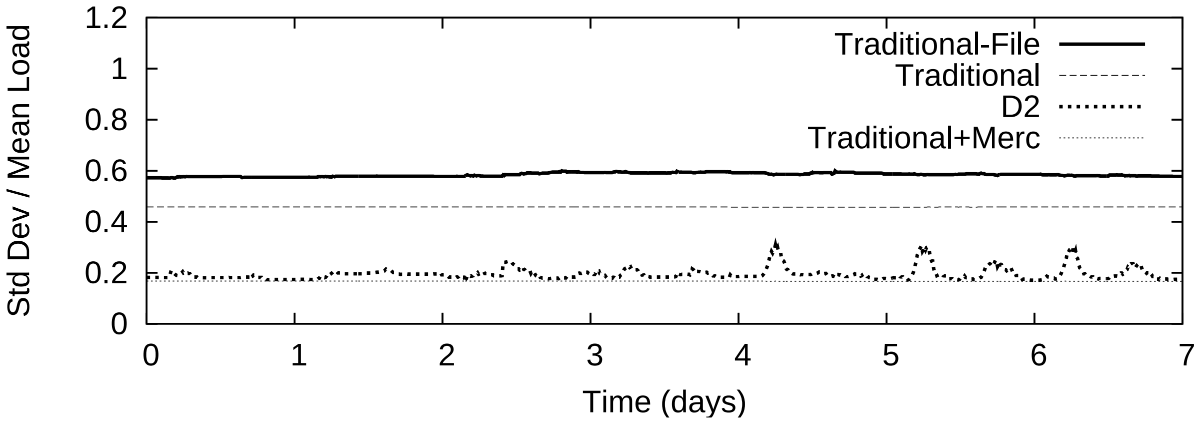


Figure 16: Load imbalance over time with `Harvard`.

distribution changes rapidly. Table 3 shows the amount of data written and removed during days 1 to 6 in each workloads, relative to the total amount present in the system at the start of each day. 10-20% of the data in the `Harvard` workload is new and an approximately equal amount is removed each day,[10] whereas the `Webcache` workload writes as much as 1330% more than existing data in a single day and all data present at the start of each day is removed by the end (ratios are larger than 1 since some data is written and removed on the same day).

Load Balance. Figure 16 and Figure 17 plot how load imbalance fluctuates over time in a traditional-file DHT, a traditional DHT, D2, and a traditional DHT that also uses Mercury's active load balancing (Traditional+Merc). We measure load imbalance with the normalized standard deviation of total node storage load (i.e., the standard deviation divided by the mean). This metric captures the deviation of node loads from the mean load (the per-node load in a perfectly balanced system).

For the `Harvard` workload, D2 is able to keep imbalance even lower than that achieved by the traditional DHT. The traditional-file DHT has the worst load balance because nodes that store larger files have higher load and the difference between the mean and maximum file size in this workload is over 4 orders of magnitude. The Traditional+Merc line shows the load balance achiev-

[10]Data churn is lower in reality since the our initial file system does not include files that are not observed in the trace. Hence, our load balancing results are actually pessimistic.

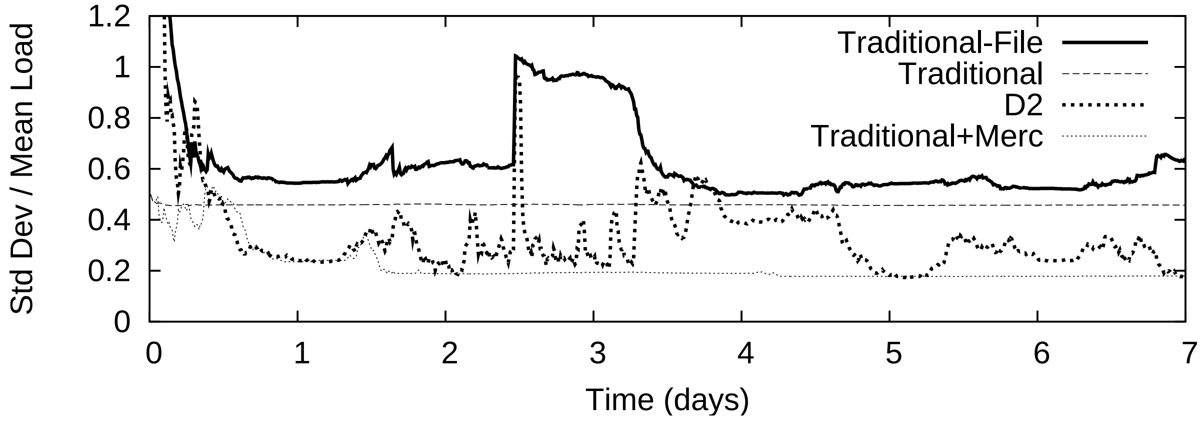


Figure 17: Load imbalance over time with `Webcache`.

Day	**1**	**2**	**3**	**4**	**5**	**6**	**Total**
`Harvard` W_i	61	71	142	114	109	123	620
`Harvard` L_i	0	18	65	60	71	93	307
`Webcache` W_i	353	32	36	398	428	355	1602
`Webcache` L_i	247	148	45	582	425	413	1860

Table 4: Mean write traffic W_i vs. mean load balancing traffic L_i on each day i (MB).

able in a system using both consistent hashing and the active load balancing algorithm described in Section 6. D2's load balance is very close to that of the Traditional+Merc system most of the time. Thus, D2 sacrifices little in terms of load balance by giving up consistent hashing.

In addition, we note that the *maximum* storage load on a node over time averages 1.6 times the mean load in D2, but averages 2.4 times the mean load in the traditional DHT (not shown). The maximum load in D2 does exceed the traditional DHT at three time instances (corresponding to the 3 highest spikes in Figure 16) due to insertions of very large files, but load balance is quickly restored and node load never exceeds 4 times the mean; recall that in our configuration, load balancing is only initiated when the ratio between two node loads is at least 4.

Even if nodes run out of storage capacity, block pointers can temporarily divert data blocks to less loaded nodes, as described in Section 6. Our results demonstrate that the distribution of block pointers is quickly rebalanced. Hence, soon after reaching capacity limits, nodes will have space to migrate blocks diverted in this fashion to their proper defragmented locations.

Load balance when using `Webcache`, shown in Figure 17, is more volatile due to the extremely high data churn rate. In addition, since the DHT is initially empty, all data is written to a small number of nodes for the first few hours. Nonetheless, except for a few short spikes after warming up (which also correspond to insertions of very large files), imbalance remains below that of the traditional DHT in terms of both the standard deviation and the maximum load.

Overhead. Table 4 compares the average amount of load balancing traffic (i.e., data migrated) per node with the average amount of data written to each node by users. File creations, modifications, and removals change the distribution of data in the system, so in the worst case, every time data is added or removed from the system, D2 might have to migrate some data to balance load. Table 4 shows that with `Harvard`, load balancing traffic is only about 50% of the total write traffic over the week. This means that for every 2 bytes written, 1 byte is migrated later. Since read traffic tends to dominate write traffic in most file systems, we expect load balancing traffic to be insubstantial.

With `Webcache`, load balancing traffic is only slightly more than the write traffic. Hence, workloads with extremely high data churn might require each byte written also to be moved once. D2 still ensures that the overhead is not much greater than the write traffic volume by postponing data migration with block pointers to avoid unnecessary transfers.

11 Conclusion and Future Work

This paper demonstrated the significant availability and performance benefits of a “defragmented” DHT-based file system. Our design of such a system, D2, contributes three key techniques: locality preserving keys, lookup caches, and low overhead load balancing. Our evaluation of D2 shows that, compared to traditional designs, D2 decreases unavailability by over an order of magnitude and improves user-perceived latency by 30–100% in a 1,000 node system. Since defragmentation is orthogonal to many other DHT techniques, including variations in object replication, lookup optimization, and administrative boundary maintenance, we believe its substantial value can complement many existing systems.

Two issues that may hinder D2 in certain environments are the subject of future work. First, when the infrastructure is not trusted, malicious nodes can take over arbitrary regions of the ID space by joining at those locations because node IDs in D2 are not secure hashes. Second, D2 would not perform as well with workloads that do not resemble traditional filesystems, such as those that mostly access very large files. We believe that a combination of locality preserving and consistent hashing replica placement could safeguard data and enable high performance operations on small and large files in these scenarios.

12 Acknowledgments

We thank our ICDCS shepherd Y. Charlie Hu, Mukesh Agrawal, David Andersen, Peter Steenkiste, and the anonymous reviewers for their suggestions, Daniel Ellard, HP Labs, and NLANR (supported by NSF grants NCR-9616602 and NCR-9521745) for providing traces, and Emulab for providing resources used in our experiments. This work is supported by NSF grant CNS-0435382 and NUS grants R-252-050-284-101 and R-252-050-284-133.

References

[1] Aditya Akella, Srinivasan Seshan, and Hari Balakrishnan. The impact of false sharing on shared congestion management. In *ICNP*, 2003.

[2] Berkeley DB. http://www.sleepycat.com/.

[3] R. Bhagwan et al. Total Recall: System support for automated availability management. In *NSDI*, 2004.

[4] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, 2004.

[5] B. Chun et al. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *SIGCOMM CCR*, 33(3), 2003.

[6] B.-G. Chun et al. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.

[7] F. Dabek, et al. Designing a DHT for low latency and high throughput. In *NSDI*, 2004.

[8] F. Dabek et al. Wide-area Cooperative Storage with CFS. In *SOSP*, 2001.

[9] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. In *FAST*, 2003.

[10] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.

[11] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[12] Harvard trace. http://www.eecs.harvard.edu/sos/traces.html.

[13] N. Harvey et al. Skipnet: A scalable overlay network with practical locality properties. In *USITS*, 2003.

[14] J. Hendricks et al. Improving small file performance in object-based storage. Technical Report CMU-PDL-06-104, Carnegie Mellon PDL, 2006.

[15] J. H. Howard et al. Scale and performance in a distributed file system. *ACM TCS*, 6(1):51–81, 1988.

[16] HP trace. http://www.hpl.hp.com/research/ssp/software/.

[17] IRCache. http://www.ircache.net/.

[18] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, July 2002.

[19] Craig Jensen. *Fragmentation: the Condition, the Cause, the Cure*. Executive Software International, Glendale, CA, 1994.

[20] M. Ji et al. Archipelago: An island-based file system for highly available and scalable internet services. In *USENIX Windows Systems Symposium*, 2000.

[21] David Karger and Matthias Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *ACM SPAA*, 2004.

[22] D. Karger et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.

[23] Krishnaram Kenthapadi and Gurmeet Singh Manku. Decentralized Algorithms Using Both Local and Random Probes for P2P Load Balancing. In *ACM SPAA*, 2005.

[24] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM TCS*, 10(1):3–25, 1992.

[25] Latencies. http://www.pdos.lcs.mit.edu/p2psim/kingdata.

[26] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient Management of DHT Routing Tables. In *NSDI*, May 2005.

[27] David Mazieres. A toolkit for user-level file systems. In *USENIX Technical Conference*, 2001.

[28] Marshall K. McKusick et al. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[29] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of the USENIX Winter Technical Conference*, 1991.

[30] Mercury. http://www.cs.cmu.edu/∼ashu/gamearch.html.

[31] A. Mislove and P. Druschel. Providing administrative control and autonomy in peer-to-peer overlays. In *IPTPS*, San Diego, CA, February 2004.

[32] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. In *OSDI*, 2002.

[33] Erich M. Nahum, Marcel Rosu, Srini Seshan, and Jussara Almeida. The effects of wide-area conditions on www server performance. In *SIGMETRICS*, Cambridge, MA, June 2001.

[34] J. Pang et al. Defragmenting DHT-based Distributed File Systems. Technical Report CMU-CS-07-115, Carnegie Mellon, 2007.

[35] PlanetLab Data. http://www.pdos.lcs.mit.edu/∼strib/pl_app/.

[36] S. Rhea et al. Pond: the OceanStore Prototype. In *USENIX FAST*, March 2003.

[37] Rhea, S., et al. Open DHT: A Public DHT Service and Its Uses. In *SIGCOMM*, 2005.

[38] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale persistent peer-to-peer storge utility. In *SOSP*, 2001.

[39] B. White et al. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.

[40] H. Yu, P. B. Gibbons, and S. K. Nath. Availability of Multi-Object Operations. In *NSDI*, 2006.