Defragmenting DHT-based Distributed File Systems

Jeffrey Pang* Phillip B. Gibbons† Michael Kaminsky† Srinivasan Seshan* Haifeng Yu‡
*Carnegie Mellon University †Intel Research Pittsburgh ‡National University of Singapore

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 [2, 7, 10, 27, 31] rely on *consistent hashing* [20] 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 [7], 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 [32], 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. Hereafter, 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. Can 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 contribute three techniques that enable us

to answer each question affirmatively: *locality preserving keys*, *lookup caches*, and *block pointers* with active load balancing. In addition to showing the value of these techniques in a defragmented DHT-based file system, we conduct an extensive evaluation on Emulab [33] 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 [9, 23, 24] and defragmentation [17] 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 [14] clusters files into volumes to improve system operability and Archipelago [18] 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.

DHTs and Availability. Numerous DHT-based file systems have been proposed, including CFS [7], Ivy [27], Pond [31], PAST [32], Total Recall [2], and Glacier [10]. 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 [2, 5, 10] (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.* [34]. 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.

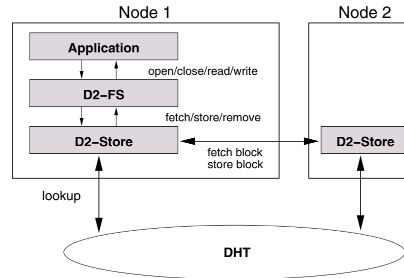


Figure 1. D2 architecture.

Administrative Locality. Mislove *et al.* [26] and Harvey *et al.* [12] 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 [3, 19] 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 CFS [7]. 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. In this organization, each block is identified by a distinct DHT key. D2 organizes these keys to preserve locality. In CFS, keys are content-hashes and are used to verify the integrity of blocks as well as to locate them. Keys in D2-FS are no longer content hashes so each pointer in D2-FS — such as a file pointer in a directory block — is paired with the content hash of the block it points to, in order to enable integrity verification.

To avoid excessive overhead on writes, D2-FS maintains a 30-second write-back cache, which is also used as a buffer cache for reads. 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.

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. 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 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 demon-

Table 1. Workloads analyzed.

Workload	Duration	Accesses	Active Data	Description
`HP` (1999) [15]	1 week	238M	40GB	A block-level trace from a multi-disk research server.
`Harvard` (2003) [11]	1 week	60M	83GB	Research and email NFS trace. (EECS workload from [8].)
`Web` (2003) [16]	1 week	47M	93GB	Accesses to web sites seen by NLANR Web caches.

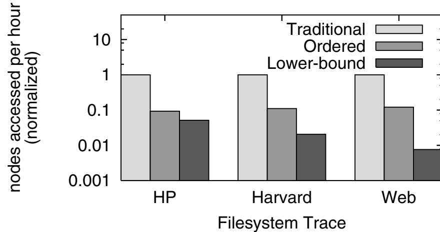


Figure 2. Mean nodes accessed per user each hour, normalized against Traditional.

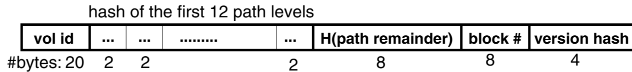


Figure 3. Key encoding for blocks in D2-FS.

strate D2's effectiveness despite the small load imbalance and temporary fragmentation that occurs when using D2's actual load balancing algorithm.

Figure 2 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 3, 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.[1] 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 [13] can be used to make it more compact.

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.

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.

[1] 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.

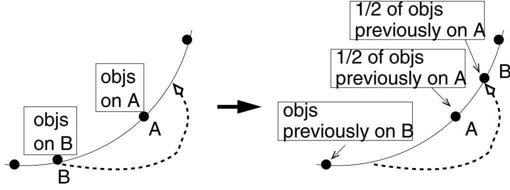


Figure 4. Load balancing example.

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 [4] (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 [3, 19]. These DHTs were originally designed to support range queries, but D2 uses their load balancing 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 4). 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 [19]. 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 [19]; 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 [3], 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.[2]

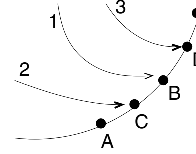


Figure 5. Example of unnecessary data transfers during load balancing (see text).

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 [32], 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 5, 1). A may still be heavily loaded (and now B as well), so later C (Figure 5, 2) and D (Figure 5, 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, 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*.

7. Prototype Implementation

D2 is implemented in C++ and uses libasync [22] for asynchronous event processing. D2-Store stores data blocks in BerkeleyDB [1] and uses Mercury [25] 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. The implementation consists of about 13K lines of code.

[2]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 $\frac{r \cdot max}{r \cdot min} = c$.

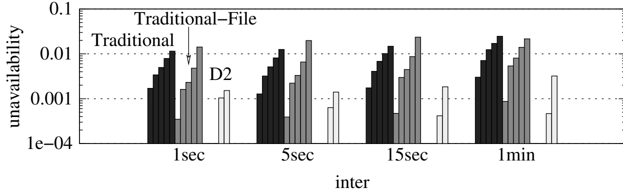


Figure 6. Task unavailability under each system while varying $inter$.

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 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.Therefore, 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 [4] from February 22 to 28, 2003, a week with a particularly large number of failures (see [30] 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.

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

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 6 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.

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 p^{23}.[3] 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 $1-p^4$). Since 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

This section evaluates how defragmentation affects 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 [33] 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 [28]) 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 [21]. 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 [4], 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.

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. All 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 [6, 7]). To make our experiments tractable, we evaluate system performance during 8 representative 15-minute periods of the trace, comprising the access patterns of 83 users.

[3]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.

Further details about our Emulab setup can be found in an associated technical report [29].

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 initiated 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. 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.

Sequential Performance. Figure 7 shows the average speedup of D2 over a traditional DHT with different sys-

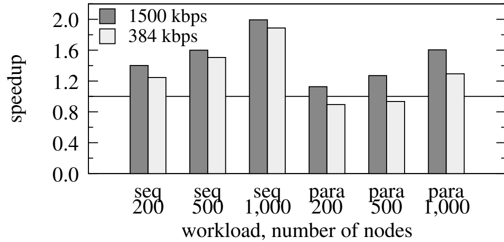


Figure 7. Speedup over a traditional DHT.

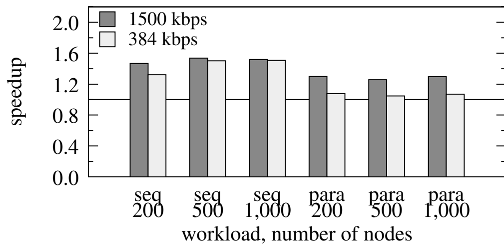


Figure 8. Speedup over a traditional-file DHT.

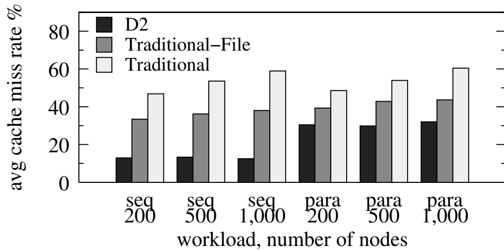
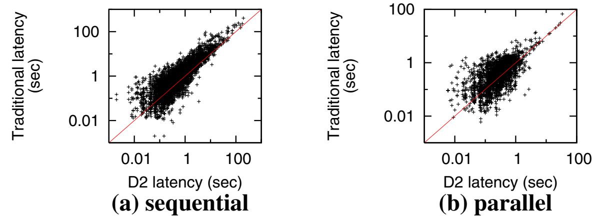


Figure 9. Mean lookup cache miss rate.

tem 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 1000, the performance improvement is at least 90%. Figure 8 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).

The lookup cache miss rate is primarily responsible for the speedups we observe. Figure 9 shows the average per-user lookup cache miss rates of each system in each of Figure 7's scenarios. In the **seq** cases, D2 has miss rates of 13% while the traditional design has miss rates of more than 47%. Moreover, the miss rate for D2 is independent of system size, while the miss rate in a traditional DHT grows with system size. D2's effective utilization of the cache also reduces the number of lookup messages by an order of magnitude when compared to a traditional DHT [29]. 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.



(a) sequential **(b) parallel**

Figure 10. Comparison of access group latencies for D2 and the traditional DHT. Note the logarithmic scales.

Finally, Figure 10(a) compares the latency of access groups under D2 and the traditional DHT in the 1000 node, 1500kbps scenario. Results comparing D2 with a traditional-file DHT are similar [29]. 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 order of magnitude.

Parallel Performance. Figure 7 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.

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 8 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 be-

cause 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 9), 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 10(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 reenters 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. 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 [29], 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 [6] are designed to avoid this adverse interaction between large parallel downloads and TCP by avoiding per-flow slow-start. However, 86% of access groups in our workload access at most 15 blocks and would not benefit from STP because our traditional DHT already fetches 15 blocks at once. Therefore, even using proposed custom transfer protocols would not substantially improve the traditional DHT's overall parallel download performance in this scenario.

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?

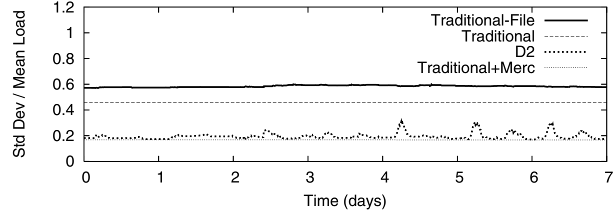


Figure 11. Load imbalance over time.

Table 3. Mean write traffic W_i vs. mean load balancing traffic L_i on each day i (in MB).

Day	1	2	3	4	5	6	Total
W_i	61	71	142	114	109	123	620
L_i	0	18	65	60	71	93	307

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 an associated technical report [29], we also show that D2 can maintain load balance and limit overhead under the extreme conditions imposed by a web cache workload, which has much more churn in its data distribution.

Load Balance. Figure 11 plots 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 achievable 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. Even when too much data is inserted into a few nodes, as exemplified by the spikes on days 4, 5, and 6, the distribution of blocks is quickly rebalanced in D2.

Overhead. Table 10 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 10 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.

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.

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 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] Berkeley DB. http://www.sleepycat.com/.
[2] R. Bhagwan et al. Total Recall: System support for automated availability management. In *NSDI*, 2004.
[3] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, 2004.
[4] B. Chun et al. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *SIGCOMM CCR*, 33(3), 2003.
[5] B.-G. Chun et al. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.
[6] F. Dabek, et al. Designing a DHT for low latency and high throughput. In *NSDI*, 2004.
[7] F. Dabek et al. Wide-area Cooperative Storage with CFS. In *SOSP*, 2001.
[8] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *FAST*, 2003.
[9] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.
[10] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.
[11] Harvard trace. http://www.eecs.harvard.edu/sos/traces.html.
[12] N. Harvey et al. Skipnet: A scalable overlay network with practical locality properties. In *USITS*, 2003.
[13] J. Hendricks et al. Improving small file performance in object-based storage. Technical Report CMU-PDL-06-104, Carnegie Mellon PDL, 2006.
[14] J. H. Howard et al. Scale and performance in a distributed file system. *ACM TCS*, 6(1):51–81, 1988.
[15] HP trace. http://www.hpl.hp.com/research/ssp/software/.
[16] IRCache. http://www.ircache.net/.
[17] C. Jensen. *Fragmentation: the Condition, the Cause, the Cure*. Executive Software International, Glendale, CA, 1994.
[18] M. Ji et al. Archipelago: An island-based file system for highly available and scalable internet services. In *USENIX Windows Systems Symposium*, 2000.
[19] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *ACM SPAA*, 2004.
[20] D. Karger et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.
[21] Latencies. http://www.pdos.lcs.mit.edu/p2psim/kingdata.
[22] D. Mazieres. A toolkit for user-level file systems. In *USENIX Technical Conference*, 2001.
[23] M. K. McKusick et al. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
[24] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of the USENIX Winter Technical Conference*, 1991.
[25] Mercury. http://www.cs.cmu.edu/∼ashu/gamearch.html.
[26] A. Mislove and P. Druschel. Providing administrative control and autonomy in peer-to-peer overlays. In *IPTPS*, San Diego, CA, Feb. 2004.
[27] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *OSDI*, 2002.
[28] E. M. Nahum, M. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on www server performance. In *SIGMETRICS*, Cambridge, MA, June 2001.
[29] J. Pang et al. Defragmenting DHT-based Distributed File Systems. Technical Report CMU-CS-07-115, Carnegie Mellon, 2007.
[30] PlanetLab Data. http://www.pdos.lcs.mit.edu/∼strib/pl_app/.
[31] S. Rhea et al. Pond: the OceanStore Prototype. In *USENIX FAST*, March 2003.
[32] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale persistent peer-to-peer storge utility. In *SOSP*, 2001.
[33] B. White et al. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.
[34] H. Yu, P. B. Gibbons, and S. K. Nath. Availability of Multi-Object Operations. In *NSDI*, 2006.