UsenetDHT: A low-overhead design for Usenet

Emil Sit, Robert Morris, and M. Frans Kaashoek
MIT CSAIL

Abstract

Usenet is a popular distributed messaging and file sharing service: servers in Usenet flood articles over an overlay network to fully replicate articles across all servers. However, replication of Usenet's full content requires that each server pay the cost of receiving (and storing) over 1 Tbyte/day. This paper presents the design and implementation of UsenetDHT, a Usenet system that allows a set of cooperating sites to keep a shared, distributed copy of Usenet articles. UsenetDHT consists of client-facing Usenet NNTP front-ends and a distributed hash table (DHT) that provides shared storage of articles across the wide area. This design allows participating sites to partition the storage burden, rather than replicating all Usenet articles at all sites.

UsenetDHT requires a DHT that maintains durability despite transient and permanent failures, and provides high storage performance. These goals can be difficult to provide simultaneously: even in the absence of failures, verifying adequate replication levels of large numbers of objects can be resource intensive, and interfere with normal operations. This paper introduces Passing Tone, a new replica maintenance algorithm for DHash [7] that minimizes the impact of monitoring replication levels on memory and disk resources by operating with only pairwise communication. Passing Tone's implementation provides performance by using data structures that avoid disk accesses and enable batch operations.

Microbenchmarks over a local gigabit network demonstrate that the total system throughput scales linearly as servers are added, providing 5.7 Mbyte/s of write bandwidth and 7 Mbyte/s of read bandwidth per server. UsenetDHT is currently deployed on a 12-server network at 7 sites running Passing Tone over the wide-area: this network supports our research laboratory's live 2.5 Mbyte/s Usenet feed and 30.6 Mbyte/s of synthetic read traffic. These results suggest a DHT-based design may be a viable way to redesign Usenet and globally reduce costs.

1 Introduction

For decades, the Usenet service has connected users world-wide. Users post articles into newsgroups which are propagated widely by an overlay network of servers. Users host lively discussions in newsgroups and also, because articles can represent multi-media files, cooperatively produce a large shared pool of files. A major attraction of Usenet is the incredible diversity and volume of content that is available.

Usenet is highly popular and continues to grow: one Usenet provider sees upwards of 40,000 readers reading at an aggregate 20 Gbit/s [35]. Several properties contribute to Usenet's popularity. Because Usenet's design [1, 19] aims to replicate all articles to all interested servers, any Usenet user can publish highly popular content without the need to personally provide a server and bandwidth. Usenet's maturity also means that advanced user interfaces exist, optimized for reading threaded discussions or streamlining bulk downloads. However, providing Usenet service can be expensive: users post over 1 Tbyte/day of new content that must be replicated and stored.

This paper presents UsenetDHT, a system that allows a group of cooperating servers to share the network and storage costs of providing Usenet service. Such an approach benefits both operators and users of Usenet: operators can use the savings from UsenetDHT to provide better service (for example, save articles for longer), or pass on reduced costs to users. With costs lower, more organizations may be able to afford to provide Usenet and make Usenet available to more users.

UsenetDHT uses a distributed hash table (DHT) to store and maintain Usenet articles across participating sites [32]. A DHT provides a single logical storage system for all sites; it handles data placement, load balance and replica maintenance. Front-ends speaking the standard news protocol accept new articles and store them into the common DHT. The front-ends then flood information about the existence of each article to all other front-ends in the UsenetDHT deployment. Instead of transmitting and storing n copies of an article (one per server), UsenetDHT initially stores only two for the entire deployment; the extra copy allows for recovery from failure. Thus, per server, UsenetDHT reduces the load of receiving and storing articles by a factor of $O(n)$.

To service reads, UsenetDHT front-ends must read from the DHT. The front-ends employ caching to ensure that each article is retrieved at most once for local clients. Thus, even if clients read all articles, UsenetDHT never creates more copies than Usenet. Further, statistics from news servers at MIT indicate that, in aggregate, clients read less than 1% of the articles received [29, 36]. If MIT

This research was supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660, `http://project-iris.net/`.

partnered with similar institutions, UsenetDHT could reduce its costs for supporting Usenet substantially.

In addition to delivering cost savings, UsenetDHT is incrementally deployable and scalable. UsenetDHT preserves the NNTP client interface, allowing it to be deployed transparently without requiring that clients change their software. The use of an underlying DHT allows additional storage and front-end servers to be easily added, with the DHT handling the problem of re-balancing load. Additionally, UsenetDHT preserves the existing economic model of Usenet. Compared to alternative content distribution methods such as CDNs or `ftp` mirror servers that put the burden of work and cost on the publisher, UsenetDHT leaves the burden of providing bandwidth at the Usenet front-end, close to the consumer.

Despite the advantages of the UsenetDHT design, the workload of Usenet places significant demands on a DHT. The DHT must support sustained high throughput writes of multiple megabytes per second. Further, since even MIT's fractional feed of 2.5 Mbyte/s generates 350 Gbyte/day of replicas, unless a new disk is acquired every day, the system will run continuously at disk capacity, necessitating continuous object deletions to reclaim space.

The DHT that UsenetDHT uses must also provide data availability and durability. This is achieved through replica maintenance, where a DHT replaces lost replicas to prevent data loss or unavailability. The goal of replica maintenance is to avoid losing the last replica, but without making replicas unnecessarily, since objects are expensive to copy across the network. To achieve this, a maintenance algorithm must have an accurate estimate of the number of replicas of each object in order to avoid losing the last one. Since DHTs partition responsibility for maintenance across servers, a simple solution would have each DHT server check periodically with other servers to determine how many replicas exist for objects in its partition. However, naïvely exchanging lists of object identifiers in order to see which replicas are where can quickly become expensive [16].

A more efficient mechanism would be to simply exchange lists once and then track updates: a synchronization protocol (e.g., [4, 22]) can identify new objects that were added since the last exchange as well as objects that have been deleted to reclaim space. However, this alternative means that each server must remember the set of replicas held on each remote server that it synchronizes with. Further, each synchronization must cover objects inserted in well-defined periods so that when a server traverses its local view of replicas periodically to make repair decisions, it considers the same set of objects across all servers. These problems, while solvable, add complexity to the system. The storage and traversal of such sets can also cause disk activity or memory pressure, interfering with regular operations.

To handle maintenance under the workload of Usenet, we introduce *Passing Tone*, a maintenance algorithm built on Chord and DHash [7]. Instead having each object's successor ensure that sufficient replicas exist within the successor list, Passing Tone has all servers in the object's successor list ensure that they replicate the object. Each Passing Tone server makes maintenance decisions by synchronizing alternately with its successor and predecessor against the objects it stores locally. Synchronization identifies objects that the server should replicate but doesn't. These objects are pulled from its neighbors. In this way, Passing Tone ensures that objects are replicated on the successor list without having to explicitly track/count replicas, and without having to consider a consistent set of objects across several servers. As a result, Passing Tone can efficiently maintain replicas in a system where constant writes produce large numbers of objects and the need to continuously delete expired ones.

Our implementation of Passing Tone lays out data structures on disk efficiently to make writes, deletions, and synchronization operations efficient. We deployed this implementation on 12 wide-area servers, and this deployment handles the live 2.5 Mbyte/s Usenet feed received at MIT. The deployment can support an aggregate read throughput of 30.6 Mbyte/s from wide-area clients. Benchmarks run over our local gigabit network show that the total read and write capacity scale as servers are added.

This paper makes three contributions: it presents the design of UsenetDHT, a system that reduces the individual cost of operating a Usenet server for n participants by a factor of $O(n)$ through the use of a DHT; it introduces the Passing Tone algorithm that provides efficient maintenance for DHTs for workloads with many concurrent write and expiration operations; and, it demonstrates that an implementation of UsenetDHT can support MIT's Usenet feed and should scale to the full feed.

The rest of this paper is structured as follows. Section 2 describes Usenet and briefly characterizes its traffic load. UsenetDHT's design is presented in Section 3. Section 4 describes the Passing Tone algorithm. Section 5 discusses the Passing Tone implementation, which is then evaluated in Section 6. Finally, Section 7 presents related work and we conclude in Section 8.

2 Usenet background

Usenet is a popular distributed messaging service that has been operational since 1981. Over the years, its use and implementation has evolved. Now people use Usenet in two primary ways. First, users participate in discussions about specific topics, which are organized in newsgroups. The amount of traffic in these text groups has been relatively stable over the past years. Second, users post binary articles (encoded versions of pictures, audio

files, and movies). This traffic is increasing rapidly because Usenet provides an efficient way for users to distribute large multi-media files. Users can upload articles to Usenet once and then Usenet takes charge of distributing the articles.

Usenet distributes articles using an overlay network of servers that are connected in a peer-to-peer topology. Servers are distributed world-wide and each server peers with its neighbors to replicate all articles that are posted to Usenet. The servers employ a flood-fill algorithm using the NetNews Transfer Protocol (NNTP) to ensure that all articles reach all servers [1, 19].

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

The size of Usenet is hard to measure as each site sees a different amount of traffic, based on its peering arrangements. An estimate from 1993 showed an annual 67% growth in traffic [34]. Currently, in order to avoid missing articles, top servers have multiple overlapping feeds, receiving up to 3 Tbyte of traffic per day from peers, of which approximately 1.5 Tbyte is new content [10, 12]. Growth has largely been driven by increased postings of binary articles. The volume of text articles has remained relatively stable for the past few years at approximately 1 Gbyte of new text data, from approximately 400,000 articles [14]. As a result of the large volume of traffic, providers capable of supporting a full Usenet feed have become specialized. Top providers such as `usenetserver.com` and GigaNews have dedicated, multi-homed data centers with many servers dedicated to storing and serving Usenet articles.

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

The read workload at major news servers can be extremely high: on a weekend in September 2007, the popular `usenetserver.com` served an average of over 40,000 concurrent clients with an aggregate bandwidth of 20 Gbit/s [35]. This suggests that clients are downloading continuously at an average of 520 Kbit/s, most likely streaming from binary newsgroups.

Usenet's economics are structured to allow providers to handle the high costs associated with receiving, storing, and serving articles. Perhaps surprisingly in the age of peer-to-peer file sharing, Usenet customers are willing to pay for access to the content on Usenet. Large providers are able to charge customers in order to cover their costs and produce a profit. Unfortunately, universities and other smaller institutions may find it difficult to bear the cost of operating an entire Usenet feed. UsenetDHT is an approach to bring these costs down and allow more sites to operate Usenet servers.

3 UsenetDHT design

UsenetDHT targets mutually trusting organizations that can cooperate to share storage and network load. Prime examples of such organizations are universities, such as those on Internet2, that share high-bandwidth connectivity internally and whose commercial connectivity is more expensive. For such organizations, UsenetDHT aims to:

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

UsenetDHT accomplishes these goals by replacing the local article storage at each NNTP server with shared storage provided by a distributed hash table (DHT). A front-end that speaks the standard client transfer protocol (NNTP) allows unmodified clients access to this storage at each site.

3.1 Design overview

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

In UsenetDHT, each site contributes one or more servers with dedicated storage to form a DHT, which acts like a virtual shared disk. The DHT relieves UsenetDHT from solving the problem of providing robust storage; DHT algorithms deal with problems of data placement, maintenance in the face of failures, and load balance as the number of servers and objects in the system

increases [5, 33]. To support a full feed, each server in a homogeneous deployment need provide only $O(1/n)$-th of the storage it would need to support a full feed by itself.

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

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

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

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

3.2 Write and read walk-through

To demonstrate the flow of articles in UsenetDHT more precisely, this section traces the posting and reading of an article. Figure 1 summarizes this process.

A news reader posts an article using the standard NNTP protocol, contacting a local NNTP front-end. The reader is unaware that the front-end is part of UsenetDHT, and not a standard Usenet server. Upon receiving the posting, the UsenetDHT front-end uses `put` to insert the article in the DHT. In the `put` call, the front-end uses the SHA-1

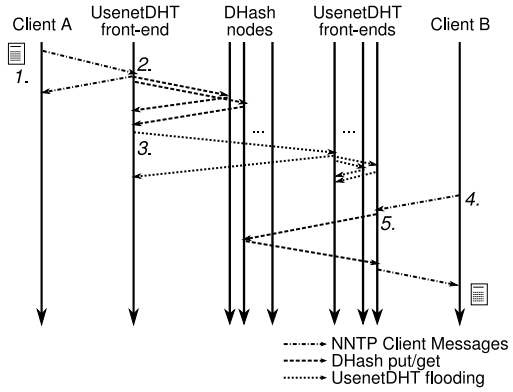

Figure 1: Messages exchanged during during UsenetDHT reads and writes. 1. Client A posts an article to his local NNTP front-end. 2. The front-end stores the article in DHash (via a DHash gateway, not shown). 3. After successful writes, the front-end propagates the article's metadata to other front-ends.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

hash of the article's content as the key: since DHash partitions data across servers by the key, using a hash function ensures articles will be distributed evenly across the participating servers. By providing this key in a `get` call, any UsenetDHT front-end can retrieve the article from the DHT. The use of a content-hash key also allows front-ends to detect data corruption by verifying the integrity of data received over the network.

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

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

from the DHT, it inserts the article into the cache and returns the article to the reader. As with posting, the reader is unaware that the news server is part of UsenetDHT.

3.3 Expiration

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

Usenet servers have long used *expiration times* to bound the lifetime of articles. Each deployment of UsenetDHT sets a single common expiration policy across all participants. This policy can vary according to the type of newsgroup (e.g., preserving text discussions for longer than binary articles). A common policy is required to ensure that the list of articles for a given group at any UsenetDHT front-end will accurately reflect the articles that are available in the underlying DHT.

3.4 Discussion

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

A DHT-based design also represents a sacrifice in terms of site control. Sites lose control over what articles are stored on and transit their servers. While sites retain the ability to specify which newsgroups are indexed locally, they must participate fully in the DHT and store articles that are in groups that they do not make available to local clients. Policies of content filtration, handled in regular Usenet by filtering the list of newsgroups that are peered to a site, must now be handled per deployment. Future work may address this issue with the use of sub-rings [20].

The key benefit of the DHT approach is that each site will receive article data proportional to the amount of storage they contribute to the system, rather than a complete copy of the feed. UsenetDHT reduces the cost of receiving and storing a Usenet feed at each site by a factor of $O(n)$ (assuming equal amounts of storage contributed by each site) by eliminating the need to massively replicate articles. Our UsenetDHT implementation replicates articles for durability at two servers, so the cost of receiving and storing articles is reduced by a factor of $n/2$.

UsenetDHT employs local caching to use no more bandwidth than standard Usenet in the worst case where every article is read at every server. Fortunately, at the organizations we target, it is unlikely that all articles will be read, resulting in substantial savings per site overall. As a result, total storage can be reduced or re-purposed to provide additional retention instead of storing replicas.

The flooding network used by UsenetDHT to propagate metadata follows peering relationships established by the system administrators. A reasonable alternative may be to construct a distribution tree automatically [17]. An efficient broadcast tree would reduce link stress by ensuring that data is transmitted over each link only once. However, the metadata streams are relatively small and operators may prefer to avoid the complexity and possible fragility involved in deploying such an overlay.

4 Passing Tone

The DHT storing articles for UsenetDHT needs an efficient maintenance algorithm that provides high availability and durability for objects: maintenance ensures that a sufficient number of replicas exists to prevent object loss due to server failures. This section introduces the Passing Tone algorithm; Passing Tone maintains replicated objects through a user-specified expiration time while reducing the number of disk seeks and memory required to make maintenance decisions.*

We present Passing Tone in the context of the DHash DHT. Each server in DHash has a single database that is shared across all of its virtual servers. DHash stores object replicas initially on the first k successors of the object's key; this set of servers is called the *replica set*. Passing Tone ensures that all members of the replica set have a replica of the object, despite the fact that this set changes over time. The value of k is set system-wide and affects the ability of the system to survive simultaneous failures; the reader is referred to our prior work for more on the role of k [5]. For UsenetDHT, $k = 2$ is sufficient.

4.1 Challenges

The main challenge in the design of Passing Tone is balancing the desire to minimize bandwidth usage (e.g., by not repeatedly exchanging object identifier lists and not creating too many replicas) with the need to avoid storing and updating state about remote servers. This problem largely revolves around deciding what information to keep on each server to make identifying objects that need repair easy and efficient.

The ideal maintenance algorithm would only generate repairs for those objects that have fewer than k remaining replicas. Such an algorithm would minimize replica movement and creation. A straightforward approach for achieving this might be to track the location of all available replicas, and repair when k or fewer remain. In a peer-to-peer system, where writes are sent to individual servers without going through any central host, tracking

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

replica locations would require that each server frequently synchronize with other servers to learn about any new updates. The resulting replica counts would then periodically be checked to identify objects with too few replicas and initiate repairs.

The difficulty with this approach lies in storing state about replica locations in a format that is easy to update as objects are added and deleted but also easy to consult when making repair decisions. For ease of update and to allow efficient synchronization, information about replicas on each server needs to be kept in a per-server structure. However, to make repair decisions, each server requires a view of replicas over all servers to know how many replicas are currently available. While these views can be derived from one another, with millions of objects across dozens of servers, it is expensive to construct them on the fly. For example, Merkle tree construction is CPU intensive and would require either random disk I/O or use significant memory. Storing both views is possible but would also require random disk I/O that would compete with regular work.

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

4.2 Passing Tone overview

Passing Tone deals with these challenges by removing the need to track object locations explicitly. Instead each server in Passing Tone:

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

Passing Tone uses *Merkle trees* for synchronization [4]. As a consequence of keeping only a single Merkle tree per server, reflecting that server's actual objects, a Passing Tone server must create a replica of objects that it is missing in order to avoid repeatedly learning about that object. The count of available replicas is maintained implicitly as servers synchronize with their neighbors: when the first k servers in an object's successor list have a replica, there are at least k replicas.

Like Carbonite [5], Passing Tone uses extra replicas to mask transient failures and provide durability. Each server in Passing Tone has two maintenance responsibilities. First, it must ensure that it has replicas of objects for which it is responsible. This is its *local maintenance* responsibility. Second, it must ensure that objects it is no longer responsible for but has stored locally are offered to the new responsible server. This represents a server's *global maintenance* responsibility.

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

Figure 2: The local maintenance procedure ensures that each server n has replicas of objects for which it is responsible. The `synchronize(a,b)` method compares the local database with the partner's database to identify objects with keys in the range (a,b) that are not stored locally. Local maintenance does not delete objects locally or remotely.

4.3 Local maintenance

The local maintenance algorithm distributes the responsibility of ensuring that sufficient replicas exist to each of the servers in the current replica set. To do this without requiring coordination across all servers simultaneously, Passing Tone's local maintenance relies on an extension to Chord that allows each server to know precisely which replica sets it belongs to: Passing Tone asks Chord to maintain a *predecessor list* which tracks the first $O(\log n)$ predecessors of each server. The details of this are discussed in Section 5. Once the predecessor list is available, each server will know the range of keys it is responsible for replicating and can identify whether or not it needs to replicate an object simply by considering the key relative to its first k predecessors.

The algorithm that implements local maintenance is shown in Figure 2. Each server synchronizes with only its direct predecessor and successor, over the range of keys for which it should be holding replicas, as determined by its Chord identifier. Synchronization walks down the relevant branches of the Merkle tree to efficiently identify objects that the server is missing but that are present on its neighbor.

Merkle trees store the keys of a server in a tree with a 64-way branching factor: each node of the tree stores a hash of the concatenation of the hashes of its children. At the bottom, the leaves of the tree represent the hash of the keys of the objects themselves. When two sub-trees represent exactly the same set of keys, the hashes of the root of the sub-trees will match: the synchronization protocol can detect this efficiently and avoid further process-

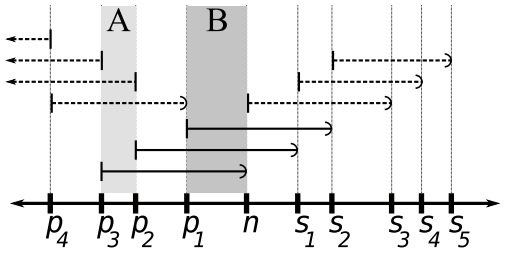

Figure 3: Server n is responsible for objects whose keys fall between its predecessor p_1 and itself. In the figure, objects are replicated with an initial replication level $k = 3$. Thus, objects that n is responsible for are replicated on s_1 and s_2. The ranges of the objects held by each server is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

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

Figure 4: Global maintenance ensures objects are placed in the correct replica sets. n periodically iterates over its database, finds the appropriate successor for objects it is not responsible for and offers them to that successor. The `reverse_sync` call identifies objects present on n but missing on s over the specified range.

ing of those sub-trees. The synchronization protocol also restricts the branches considered based on the range to be synchronized over. This allows the same tree to be used to synchronize with any server.

Any objects that synchronization identifies as missing locally are then replicated locally. By asking each server to be responsible for replicating objects to itself, Passing Tone ensures that no single server need count the number of replicas of an object. Rather, the servers in a successor list operate independently but cooperatively to ensure the right number of replicas exist.

This approach ensures that the implementation will never need to maintain or consult information from multiple servers simultaneously. This reduces the memory and disk impact of maintenance. This also avoids the problem that can lead to spurious repairs: instead of accumulating information and referring to it after it has become stale, each server in Passing Tone make decisions immediately upon synchronizing with its neighbor.

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

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

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

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

Despite only local knowledge, local maintenance does not result in excessive creation of replicas. Temporary failures do not cause replicas to be repeatedly created and then deleted because Passing Tone allows objects to remain on servers even if there are already k replicas; objects are created once and "re-used" on subsequent failures. Our first maintenance algorithm [6] kept replicas on *exactly* the first k successors, which was shown to be a bad decision [5].

4.4 Global maintenance

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

Global and local maintenance in Passing Tone are cooperative and complementary: global maintenance explicitly excludes those objects that are handled by local mainte-

nance. At the same time, global maintenance in Passing Tone relies on local maintenance to correctly propagate replicas to other servers in the official replica set. That is, once an object has been moved to the successor, the neighbors of the successor will take responsibility for creating replicas of that object across the replica set.

Figure 4 shows the pseudo-code for global maintenance. Like local maintenance, global maintenance requires the predecessor list to determine which objects each server should maintain. Server *n* periodically iterates over its database, identifying objects for which it is not in the replica set. For each such object, it looks up the current successor *s* and synchronizes with *s* over the range of objects whose keys fall in *s*'s range of responsibility. *s* then makes a copy of any objects that it is responsible for which it does not have a replica. Once these transfers are complete, global maintenance moves on to consider the next set of objects.

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

4.5 Supporting expiration

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

Failure to account for expiration in maintenance can lead to two problems. First, repairing these objects are a waste of system resources, since expired objects are ones that the application has specified as no longer requiring durability. Second, when expired objects are not expired simultaneously across multiple servers, spurious repairs can occur if one server chooses to delete an expired object before its neighbor: the next local maintenance round could identify and re-replicate that object.

One solution to this problem would be to include extra metadata during synchronization so that repairs can be prioritized based on expiration time. Including the metadata initially seems attractive: especially when interacting with mutable objects, metadata may be useful for propagating updates in addition to allowing expiration-prioritized updates. OpenDHT implements a simple variant of this by directly encoding the insertion time into the Merkle tree [27]; this complicates the construction of Merkle trees however and requires a custom Merkle tree be constructed per neighbor.

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

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

Process	kLoC	Role
`usenetdht`	3.4	Usenet front-end and DHash interface.
`lsd`	28.4	Chord routing, DHash gateway and server logic.
`adbd`	1.5	Main data storage.
`maintd`	6.8	Maintenance including Passing Tone and Merkle synchronization.

Table 1: UsenetDHT and DHT processes breakdown.

5 Implementation

This section provides an overview of our DHash, Passing Tone and UsenetDHT implementations. We highlight the particular problems and lessons that we learned while implementing these systems to meet our performance requirements.

5.1 Source code

DHash, Passing Tone and UsenetDHT are implemented in C++ using the `libasync` libraries [21]. Each host runs four separate processes, detailed in Table 1.

`usenetdht` implements NNTP to accept feeds and interact with Usenet clients. It stores each article as a single DHash content-hash object; use of a single object minimizes the overhead incurred by DHash during storage, relative to chunking objects into, for example, 8 Kbyte blocks. `usenetdht` maintains a database containing the overview metadata and content-hash for each article, organized by newsgroup.

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

The DHash and Passing Tone implementation is structured into three processes to avoid having disk I/O calls block and delay responses to unrelated network requests. Network I/O and the main logic of Chord and DHash are handled by `lsd`. Disk I/O is delegated to `adbd`, which

handles storage, and `maintd`, which implements Passing Tone and handles maintenance.

The complete DHash source base consists of approximately 38,000 lines of C++. This includes the Chord routing layer and research code for previous DHT and maintenance algorithms. The Merkle synchronization subsystem consists of approximately 3,000 lines of code; the Passing Tone implementation and supporting infrastructure is less than 1,000 lines of code.

5.2 Predecessor lists

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

Predecessor lists are not guaranteed to be correct. Servers are not considered fully joined into the Chord ring until they have received a notification from their predecessor. The predecessor maintenance algorithm may return a shortened, wrong or empty predecessor list in this situation. For this reason, this algorithm was rejected in the design of Koorde [18], which relies on predecessor lists for correct routing. However, this situation occurs fairly rarely in practice, if at all, and we have never observed any problems resulting from this implementation. Because Passing Tone does not rely on predecessor lists for routing, we accept the occasional possibility of error. Even in the event that an incorrect predecessor list is used for determining maintenance, Chord stabilization will soon cause the predecessor list to change and any outstanding incorrect repairs will be flushed before significant bandwidth has been used.

5.3 Load balance

DHash uses virtual servers for load balance [6]. To ensure that replicas are placed on virtual servers of different physical servers, `lsd` filters co-located virtual servers from the successor list during write operations, similar to Y0 [13]. Co-located virtual servers are similarly filtered from successor (and predecessor) lists during read and maintenance. However, currently, administrators must manually balance the number of virtual servers used so that each physical servers does not receive more load than its network link or disk can support. We know of no algorithm that balances load over multiple constraints.

5.4 Expiration support

To support expiration in UsenetDHT, DHash now allows applications to provide a per-object expiration time. DHash treats this time as a recommendation: DHash servers stop performing maintenance of expired objects but only delete the objects when storage is needed for new writes. In this way, the application maintains control of how long data is durably maintained, and DHash servers can operate without consulting the application in prioritizing objects for maintenance. The implementation of expiration in DHash is inspired by the timehash system used by INN (described in [31]).

5.5 Storage performance

`adbd` stores objects and metadata separately. It stores objects in append-only flat files, and stores metadata, including the Merkle synchronization tree, using BerkeleyDB databases. BerkeleyDB databases provide a simple key-value store, here used to map object keys to metadata. Storing objects outside of BerkeleyDB is important for performance. Since BerkeleyDB stores data and key within the same BTree page and since pages are fixed size, objects larger than a single page are stored on separate overflow pages. This results in particularly poor disk performance as overflow pages cause writes to be chunked into page-sized fragments that are not necessarily located contiguously, leading to additional disk seeks.

`adbd` names object storage files by approximate expiration time. Each file holds 256 seconds worth of objects, which works well for a system with high write workload. Files can be appended to efficiently, and object data can be read without extra seeks. Grouping multiple objects into a single file allows efficient reclamation of expired data when the disk approaches capacity: 256 seconds worth of objects can be freed by unlinking a single file.

`adbd` uses one database to map object keys to metadata such as the object's size, expiration time, and offset within the flat file storing the object data. A separate pair of databases stores information corresponding to the Merkle tree. `adbd` handles write requests and updates the Merkle tree as new objects are written and old ones are expired. `maintd` reads from this same tree during synchronization. BerkeleyDB's transaction system is used to provide atomicity, consistency and isolation between databases and between processes. For performance, the implementation disables writing the transaction log synchronously to disk.

Persistently storing the Merkle tree improves start-up performance. An earlier implementation stored object data with metadata and reconstructed an in-memory Merkle tree during start-up. BerkeleyDB could not efficiently enumerate the keys of the database; each page read returned few keys and much data. Further, BerkeleyDB does not include an API for enumerating the database in on-disk order. A straightforward reconstruction of the Merkle tree would require one seek per key over the entire set of keys held in the tree. With a 120 Gbyte disk (hold-

ing 600,000 173 Kbyte articles), at 5ms per seek, enumerating these keys could take over 50 minutes.

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

6 Evaluation

In this section, we demonstrate the following:

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

6.1 Evaluation method

We evaluate the ability of Passing Tone to provide durability by considering its behavior using a trace-driven simulator. A trace of the disk and transient failures, from March 2005 through March 2006, on the PlanetLab testbed [25] drives the simulation. The trace and simulator first appeared in prior work [5].

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

Unlike PlanetLab hosts, these machines are lightly loaded, very stable and have high disk capacity. Each machine participating in the deployment has at least a single 2.4 Ghz CPU, 1 Gbyte of RAM and UDMA133 SATA disks with at least 120 Gbyte free. These machines are not directly under our control and are shared with other users; write caching is enabled on these machines.

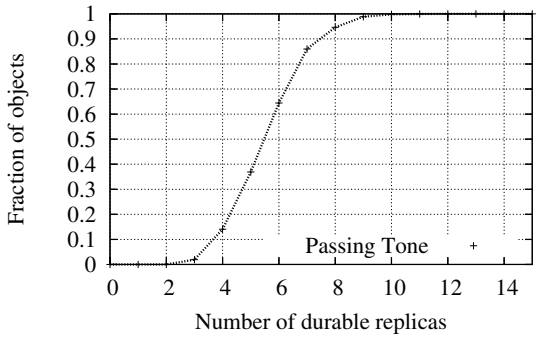

Figure 5: Durability of objects in Passing Tone: the graph shows a CDF of the number of replicas per object after a one year PlanetLab trace. No objects are lost during this simulation.

The load for the live deployment is a mirror of the CSAIL news feed; the CSAIL feed is ranked 619$^{\text{th}}$ in the Usenet Top 1000 servers [11]. This feed sees one million articles per day on average and is around 10% of the full feed. This feed generates a roughly continuous 2.5 Mbyte/s of write traffic. Binary articles have a median article size of 240 Kbyte; text articles are two orders of magnitude smaller, with a median size of 2.4 Kbyte.

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

6.2 Passing Tone durability

We evaluate Passing Tone's durability in simulation over a real-world PlanetLab trace. There are a total of 632 unique hosts experiencing 21,255 transient failures and 219 disk failures. Failures are not evenly distributed, with 466 hosts experiencing no disk failures, and 56 hosts experiencing no disk or transient failures. At the start of the trace, 50,000 20 Mbyte objects are inserted and replicated according to standard DHash placement. With an average of 490 online servers and $k = 2$ replicas, this corresponds to just over 4 Gbyte of data per server. To approximate PlanetLab bandwidth limits, each server has 150 Kbyte/s of bandwidth for object repairs. At this rate, re-creating the contents of a single server takes approximately 8 hours.

For Passing Tone to be viable, it must not lose any objects. Passing Tone synchronizes with either its predecessor or successor to generate repairs every ten minutes or when these servers change. Figure 5 shows a CDF of the number of replicas for each object at the end of the trace: the most important feature is that all objects have at least three replicas. No objects are lost, showing that Passing

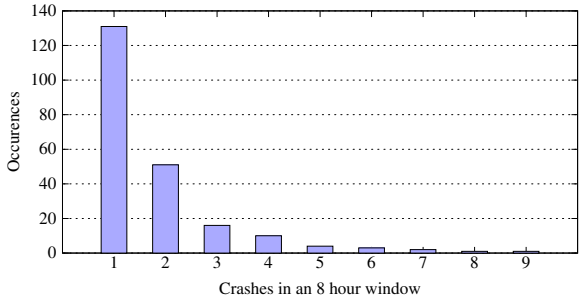

Figure 6: Number of crashes within an eight hour window over the course of the PlanetLab trace. Eight hours represents the approximate time needed to replicate the data of one PlanetLab server to another.

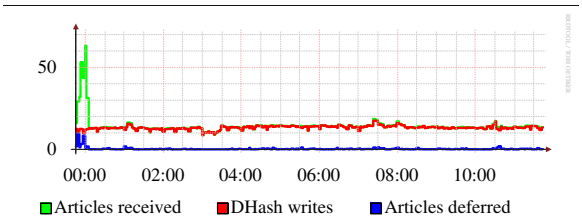

Figure 7: Articles received, posted and deferred by the CSAIL UsenetDHT deployment over a twelve hour period. During normal operation, deferrals are very rare.

Tone can provide durability in a PlanetLab environment.

The CDF also demonstrates that some objects have many more replicas; ten percent have over seven replicas. To understand why, consider Figure 6, which shows the number of times a given number of servers crashed (i.e., lost disks) within an 8 hour period over the entire trace; the worst case data loss failure in the trace could only be protected if at least nine replicas existed for objects on the nine simultaneously failing servers. It is unlikely that the actual failure in the trace would have mapped to a contiguous set of servers on a Chord ring; however, the simulation shows that Passing Tone saw sufficient transient failures to create nine replicas in at least some cases.

6.3 UsenetDHT wide-area performance

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

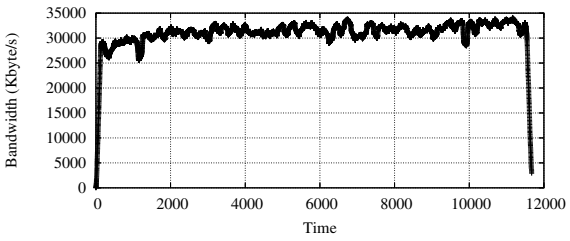

Figure 8: UsenetDHT aggregate read throughput. 100 hosts started article reads continuously against 12 servers.

Figure 7 shows the number of articles received by UsenetDHT, posted into DHash writes, and deferred. Deferrals occur when DHash's write window is full—during initial server startup, deferrals occur since there is a backlog of posts from the upstream feed that arrive faster than the local DHash server can write replicas to remote sites. The upstream later re-sends deferred posts. During normal operation, deferrals do not occur, showing that UsenetDHT and DHash can keep up with the normal workload from the CSAIL feed.

To evaluate the read capability of the system, we use clients distributed on the PlanetLab test bed. We selected 100 of the servers with the lowest load that were geographically close to one of the seven sites that were running at the time of the experiment using CoMon [24]. Each client machine ran five parallel NNTP clients to the UsenetDHT front-end closest to them and downloaded articles continuously from newsgroups chosen at random.

Figure 8 plots the achieved aggregate bandwidth over time: the client machines were able to achieve an collectively 30.6 Mbyte/s or approximately 313 Kbyte/s per client machine. This corresponds to reading on median 2612 Kbyte/s per disk. We estimate that each article read requires seven disk seeks including metadata lookup at the Usenet server, object offset lookup, opening the relevant file and its inode, indirect and double-indirect blocks, data read, and atime update. With at least 70ms of total seek time per access, this means that each disk can support no more than 14 reads per second: given an average article size of 173 Kbyte ($14 \times 173 = 2422$ Kbyte/s), this corresponds well with the observed throughput per disk.

6.4 Passing Tone efficiency

When failures do arise, Passing Tone is able to identify repairs without substantially affecting the ability of the system to process writes. Figure 9 demonstrates the behavior of Passing Tone under transient failures and server addition in a six hour window, by showing the total number of objects written: the number of objects repaired is shown in a separate color over the number of objects written. At 18:35, we simulated a 15 minute crash-and-reboot

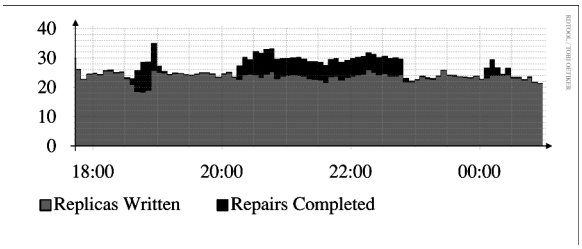

Figure 9: Number of repairs and number of writes over time, where we introduce a simple transient failure, a server addition, and a second transient failure. The second failure demonstrates behavior under recovery and also what would have happened in a permanent failure.

cycle on a server responsible for 8.5% of the data in the system. Passing Tone immediately begins repairing and ceases when that server comes back online at 18:50. A more permanent failure would involve a longer period of repairs, as repair speed is limited largely by bandwidth.

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

Finally, we demonstrate a short transient failure of the same new server at 00:05. Its neighbors begin repairing the objects that were inserted to the failed server during its four hour lifetime: however, none of the objects that were stored on the original successor needed to be repaired because its replicas were not deleted. When the new server is brought back online at 00:20, there are a few transient repairs to bring it up to date with the objects that were written during its downtime but not a repeat of the four hour initial transfer. This second transient failure also indicates how Passing Tone would behave after a permanent failure: we expect the transient failures will create a small buffer of extra replicas so that when a permanent failure does occur, only those (relatively) few objects that have been inserted since will require re-replication.

6.5 DHash microbenchmarks

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

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

Table 2: Read and write microbenchmark performance.

To test the read and write scalability of the implementation, we configure one client per DHT server and direct each client to write (and read) a 2 Gbyte synthetic stream of 200 Kbyte objects as fast as possible. Replication, maintenance and expiration are disabled. The resulting load is spread out over all the servers in the system. The results are shown in Table 2.

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

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

7 Related work

The problems of costs from repetitive article transmission and replication of un-read articles in Usenet have been noted previously. Newscaster [2] examined using IP mul-

ticast to transfer news articles to many Usenet servers at once. Each news article only has to travel over backbone links once, as long as no retransmissions are needed. In addition, news propagation times are reduced. However, Newscaster still requires that each Usenet server maintain its own local replica of all the newsgroups. IP multicast also requires extensive infrastructure within the network, which is not required by overlays such as DHash.

NewsCache [15] is one of several projects that reduce bandwidth at servers by caching news articles. It is designed to replace traditional Usenet servers that are leaf nodes in the news feed graph, allowing them to only retrieve and store articles that are requested by readers. In addition to filling the cache on demand, it can also prefetch articles for certain newsgroups. These features are also available as a mode of DNews [23], a commercial high-performance server, which adds the ability to dynamically determine the groups to pre-fetch. Both NewsCache and DNews reduce local bandwidth requirements to be proportional to readership and use local storage for caching articles of interest to local readers. UsenetDHT employs these caching strategies but also makes use of a DHT to removes the need to pay an upstream provider to source the articles.

The Coral and CobWeb CDNs provide high-bandwidth content distribution using DHT ideas [9, 37]. Coral uses distributed sloppy hash tables that optimize for locality and CobWeb locates data using Pastry [3] (with content optimally replicated using Beehive [26]). However, CDNs cache content with the aim of reducing latency and absorbing load from an origin server. UsenetDHT operates in a system without origin servers and must guarantee durability of objects stored.

Dynamo is an Amazon.com-internal DHT and uses techniques for load balance, performance and flexibility that are similar to those used in DHash and Passing Tone [8]. Dynamo is deployed in cluster-oriented environments, with fewer wide area links than DHash. OpenDHT is deployed on PlanetLab and provides a DHT for public research use [27, 28]. Compared to DHash, OpenDHT focuses on storing small data objects. Like DHash, OpenDHT provides time-to-live (TTLs) but uses them to guarantee fair access to storage across multiple applications. Both Dynamo and OpenDHT also use Merkle trees for synchronization but without explicit support for expiration.

Passing Tone optimizes maintenance for accuracy and dealing with expiration. Like Carbonite, Passing Tone keeps extra copies of objects on servers in the successor list to act as a buffer that protects the number of available objects from falling during transient failures [5]. Like Passing Tone, PAST allows more than k replicas of an object but PAST's replicas are primarily for performance not durability [30]; Passing Tone manages disk capacity with expiration, not diversion. Passing Tone's division of local and global maintenance originally was proposed by Cates [4] and has also been used in OpenDHT [27].

Despite a high number of objects, Passing Tone's use of Merkle synchronization trees [4] reduces bandwidth use during synchronization, without the complexity of aggregation as used in Glacier [16]. The idea of eventual consistency in Passing Tone is similar to Glacier's use of rotating Bloom filters for anti-entropy. However, Bloom filters still repetitively exchange information about objects and, at some scales, may be infeasible. Minsky *et al*'s synchronization algorithm [22] is more network efficient than Merkle trees but available implementations do not support persistence; in practice synchronization bandwidth is dwarfed by data transfer bandwidth.

8 Conclusions

After three decades, Usenet continues to be an important network service because of its distinct advantages over other data distribution systems. This results in over 1 Tbyte of new content posted to Usenet per day. Usenet servers have improved dramatically to carry this level of load, but the basic Usenet design hasn't changed, even though its flooding approach to distributing content is expensive. With the current design only a limited of servers can provide the full Usenet feed. We propose to exploit the recent advances in DHTs to reduce the costs of supporting Usenet, using a design that we call UsenetDHT.

UsenetDHT aggregates n servers into a DHT that stores the content of Usenet. This approach reduces the costs of storing and receiving a feed to $O(1/n)$. To enable a DHT to store as much data as Usenet generates, we developed the Passing Tone maintenance algorithm, which provides good durability and availability, keeps the memory pressure on servers low and avoids disks seeks. Experiments with a small deployment show that Passing Tone and UsenetDHT support the 2.5 Mbyte/s feed at CSAIL and should be able to scale up to the full feed by adding more servers. These results suggest that UsenetDHT may be a promising approach to evolve Usenet and to allow it to continue to grow.

Acknowledgments Frank Dabek and James Robertson are co-authors for an earlier portion of this work published at IPTPS 2004 [32]. Garrett Wollman provided the live Usenet feed used for evaluation and graciously answered many questions. The wide-area results made use of the PlanetLab and the RON test-beds; also, Magda Balazinska, Kevin Fu, Jinyang Li, and Alex Snoeren made resources at their respective universities available to us for testing. Finally, this paper improved considerably due to the comments and suggestions of the reviewers and our shepherd, Emin Gün Sirer.

References

[1] BARBER, S. Common NNTP extensions. RFC 2980, Network Working Group, Oct. 2000.

[2] BORMANN, C. The Newscaster experiment: Distributing Usenet news via many-to-more multicast. http://citeseer.nj.nec.com/251970.html.

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

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

[5] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *Proc. of the 3rd Symposium on Networked Systems Design and Implementation* (May 2006).

[6] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating System Principles* (Oct. 2001).

[7] DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. Designing a DHT for low latency and high throughput. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004).

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

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

[10] GHANE, S. Diablo statistics for spool-1t2.cs.clubint.net (TOP1000 #24). http://usenet.clubint.net/spool-1t2/stats/. Accessed 30 September 2007.

[11] GHANE, S. The official TOP1000 Usenet servers page. http://www.top1000.org/. Accessed 13 September 2007.

[12] GIGANEWS. 1 billion Usenet articles. http://www.giganews.com/blog/2007/04/1-billion-usenet-articles.html, Apr. 2007.

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

[14] GRADWELL.COM. Diablo statistics for news-peer.gradwell.net. http://news-peer.gradwell.net/. Accessed 30 September 2007.

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

[16] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005).

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

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

[19] KANTOR, B., AND LAPSLEY, P. Network news transfer protocol. RFC 977, Network Working Group, Feb. 1986.

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

[21] MAZIÈRES, D. A toolkit for user-level file systems. In *Proc. of the 2001 USENIX Annual Technical Conference* (June 2001).

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

[23] NETWIN. DNews: Unix/Windows Usenet news server software. http://netwinsite.com/dnews.htm. Accessed 9 November 2003.

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

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

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

[27] RHEA, S. *OpenDHT: A Public DHT Service*. PhD thesis, University of California, Berkeley, 2005.

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

[29] ROLFE, A. Private communication, 2007.

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

[31] SAITO, Y., MOGUL, J. C., AND VERGHESE, B. A Usenet performance study. http://www.research.digital.com/wrl/projects/newsbench/usenet.ps, Nov. 1998.

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

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

[34] SWARTZ, K. L. Forecasting disk resource requirements for a Usenet server. In *Proc. of the 7th USENIX Large Installation System Administration Conference* (1993).

[35] Network status page. http://www.usenetserver.com/en/networkstatus.php. Accessed 30 September 2007.

[36] WOLLMAN, G. Private communication, 2007.

[37] YEE JIUN SONG AND, V. R., AND SIRER, E. G. Optimal resource utilization in content distribution networks. Computing and Information Science Technical Report TR2005-2004, Cornell University, Nov. 2005.