# OpenDHT: A Public DHT Service and Its Uses

Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz,
Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu
UC Berkeley and Intel Research
opendht@opendht.org

## ABSTRACT

Large-scale distributed systems are hard to deploy, and distributed hash tables (DHTs) are no exception. To lower the barriers facing DHT-based applications, we have created a public DHT service called OpenDHT. Designing a DHT that can be widely shared, both among mutually untrusting clients and among a variety of applications, poses two distinct challenges. First, there must be adequate control over storage allocation so that greedy or malicious clients do not use more than their fair share. Second, the interface to the DHT should make it easy to write simple clients, yet be sufficiently general to meet a broad spectrum of application requirements. In this paper we describe our solutions to these design challenges. We also report our early deployment experience with OpenDHT and describe the variety of applications already using the system.

## Categories and Subject Descriptors

C.2 [**Computer Communication Networks**]: Distributed Systems

## General Terms

Algorithms, Design, Experimentation, Performance, Reliability

## Keywords

Peer-to-peer, distributed hash table, resource allocation

## 1. MOTIVATION

Large-scale distributed systems are notoriously difficult to design, implement, and debug. Consequently, there is a long history of research that aims to ease the construction of such systems by providing simple primitives on which more sophisticated functionality can be built. One such primitive is provided by distributed hash tables, or DHTs, which support a traditional hash table's simple put/get interface, but offer increased capacity and availability by partitioning the key space across a set of cooperating peers and replicating stored data.

While the DHT field is far from mature, we have learned a tremendous amount about how to design and build them over the past few years, and several well-debugged DHT implementations [1–3] are now readily available. Furthermore, several fully deployed DHT applications are now in daily use [15,20,26], and dozens more have been proposed and/or prototyped.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SIGCOMM'05,* August 21–26, 2005, Philadelphia, Pennsylvania, USA.
Copyright 2005 ACM 1-59593-009-4/05/0008 ...$5.00.

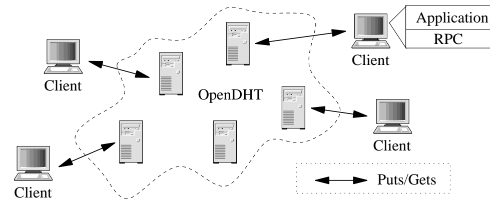


**Figure 1: OpenDHT Architecture.**

Maintaining a running DHT requires non-trivial operational effort. As DHT-based applications proliferate, a natural question to ask is whether every such application needs its own DHT deployment, or whether a shared deployment could amortize this operational effort across many different applications. While some applications do in fact make extremely sophisticated use of DHTs, many more access them through such a narrow interface that it is reasonable to expect they might benefit from a shared infrastructure.

In this paper, we report on our efforts to design and build OpenDHT (formerly named OpenHash [19]), a shared DHT deployment. Specifically, our goal is to provide a free, public DHT service that runs on PlanetLab [5] *today*. Longer-term, as we consider later in this paper, we envision that this free service could evolve into a competitive commercial market in DHT service.

Figure 1 shows the high-level architecture of OpenDHT. Infrastructure nodes run the OpenDHT server code. Clients are nodes *outside* the set of infrastructure nodes; they run application code that invokes the OpenDHT service using RPC. Besides participating in the DHT's routing and storage, each OpenDHT node also acts as a *gateway* through which it accepts RPCs from clients.

Because OpenDHT operates on a set of infrastructure nodes, no application need concern itself with DHT deployment, but neither can it run application-specific code on these infrastructure nodes. This is quite different than most other uses of DHTs, in which the DHT code is invoked as a library on each of the nodes running the application. The library approach is very flexible, as one can put application-specific functionality on each of the DHT nodes, but each application must deploy its own DHT. The service approach adopted by OpenDHT offers the opposite tradeoff: less flexibility in return for less deployment burden. OpenDHT provides a home for applications more suited to this compromise.

The service approach not only offers a different tradeoff; it also poses different design challenges. Because of its shared nature, building OpenDHT is not the same as merely deploying an existing

DHT implementation on PlanetLab. OpenDHT is shared in two different senses: there is sharing both among applications and among clients, and each raises a new design problem.

First, for OpenDHT to be shared effectively by many different applications, its interface must balance the conflicting goals of generality and ease-of-use. Generality is necessary to meet the needs of a broad spectrum of applications, but the interface should also be easy for simple clients to use. Ease-of-use argues for a fairly simple primitive, while generality (in the extreme) suggests giving raw access to the operating system (as is done in PlanetLab).[1] It is hard to quantify both ease-of-use and generality, so we rely on our early experience with OpenDHT applications to evaluate our design decisions. Not knowing what applications are likely to emerge, we can only conjecture about the required degree of generality.

Second, for OpenDHT to be shared by many mutually untrusting clients without their unduly interfering with each other, system resources must be allocated with care. While ample prior work has investigated bandwidth and CPU allocation in shared settings, storage allocation has been studied less thoroughly. In particular, there is a delicate tradeoff between fairness and flexibility: the system shouldn't unnecessarily restrict the behavior of clients by imposing arbitrary and strict quotas, but it should also ensure that all clients have access to their fair share of service. Here we can evaluate prospective designs more quantitatively, and we do so with extensive simulations.

We summarize our solutions to these two design problems in Section 2. We then address in significantly more detail the OpenDHT interface (Section 3) and storage allocation algorithm (Section 4). Section 5 describes our early deployment experience, both in terms of raw performance and availability numbers, and the variety of applications currently using the system. Section 6 concludes with a discussion of various economic concerns that may affect the design and deployment of services like OpenDHT.

# 2. OVERVIEW OF DESIGN

Before delving into the details of OpenDHT in subsequent sections, we first describe the fundamental rationale for the designs we chose for the system's interface and storage allocation mechanism.

## 2.1 Interface

In designing OpenDHT, we have the conflicting goals of generality and ease-of-use (which we also refer to as simplicity). There are three broad classes of interfaces in the DHT literature, and they each occupy very different places on the generality/simplicity spectrum (a slightly different taxonomy is described in [11]). Given a key, these interfaces provide three very different capabilities:

**routing** Provides general access to the DHT node responsible for the input key, and to each node along the DHT routing path.

**lookup** Provides general access to the DHT node responsible for the input key.

**storage** Directly supports the put(key, value) and get(key) operations by routing them to the DHT node responsible for the input key, but exposes no other interface.

The *routing* model is the most general interface of the three; a client is allowed to invoke arbitrary code at the endpoint and at every node along the DHT path towards that endpoint (either through upcalls or iterative routing). This interface has been useful in implementing DHT-based multicast [7] and anycast [34].

The *lookup* model is somewhat less general, only allowing code invocation on the endpoint. This has been used for query processing [17], file systems [9, 23], and packet forwarding [31].

The true power of the routing and lookup interfaces lies in the application-specific code running on the DHT nodes. While the DHT provides routing to the appropriate nodes, it is the application-specific code that does the real work, either at each hop en route (routing) or only at the destination (lookup). For example, such code can handle forwarding of packets (*e.g.*, multicast and *i*3 [31]) or data processing (*e.g.*, query processing).

The *storage* model is by far the least flexible, allowing no access to application-specific code and only providing the put/get primitives. This lack of flexibility greatly limits the spectrum of applications it can support, but in return this interface has two advantages: it is simple for the service to support, in that the DHT infrastructure need not deal with the vagaries of application-specific code running on each of its nodes, and it is also simple for application developers and deployers to use, freeing them from the burden of operating a DHT when all they want is a simple put/get interface.

In the design of OpenDHT, we place a high premium on simplicity. We want an infrastructure that is simple to operate, and a service that simple clients can use. Thus the storage model, with its simple put/get interface, seems most appropriate. To get around its limited functionality, we use a novel client library, Recursive Distributed Rendezvous (ReDiR), which we describe in detail in Section 3.2. ReDiR, in conjunction with OpenDHT, provides the equivalent of a lookup interface for any arbitrary set of machines (inside or outside OpenDHT itself). Thus clients using ReDiR achieve the flexibility of the lookup interface, albeit with a small loss of efficiency (which we describe later).

Our design choice reflects our priorities, but one can certainly imagine other choices. For instance, one could run a shared DHT on PlanetLab, with the DHT providing the routing service and PlanetLab allowing developers to run application-specific code on individual nodes. This would relieve these developers of operating the DHT, and still provide them with all the flexibility of the routing interface, but require careful management of the application-specific code introduced on the various PlanetLab nodes. We hope others explore this portion of the design space, but we are primarily interested in facilitating simple clients with a simple infrastructure, and so we chose a different design.

While there are no cut-and-dried metrics for simplicity and generality, early evidence suggests we have navigated the tradeoff between the two well. As we describe in greater detail in Section 5.1, OpenDHT is highly robust, and we firmly believe that the relative simplicity of the system has been essential to achieving such robustness. While generality is similarly difficult to assess, in Table 4 we offer a catalog of the diverse applications built on OpenDHT as evidence of the system's broad utility.

## 2.2 Storage Allocation

OpenDHT is essentially a public storage facility. As observed in [6, 30], if such a system offers the persistent storage semantics typical of traditional file systems, the system will eventually fill up with orphaned data. Garbage collection of this unwanted data seems difficult to do efficiently. To frame the discussion, we consider the solution to this problem proposed as part of the Palimpsest shared public storage system [30]. Palimpsest uses a novel revolving-door technique in which, when the disk is full, new stores push out the old. To keep their data in the system, clients re-put frequently enough so that it is never flushed; the required re-put rate

[1] One might argue that PlanetLab solves the problems we are posing by providing extreme resource control and a general interface. But PlanetLab is hard for simple clients to use, in that every application must install software on each host and ensure its continued operation. For many of the simple applications we describe in Section 5.3, this effort would be inappropriately burdensome.

depends on the total offered load on that storage node. Palimpsest uses per-put charging, which in this model becomes an elegantly simple form of congestion pricing to provide fairness between users (those willing to pay more get more).

While we agree with the basic premise that public storage facilities should not provide unboundedly persistent storage, we are reluctant to require clients to monitor the current offered load in order to know how often to re-put their data. This adaptive monitoring is complicated and requires that clients run continuously. In addition, Palimpsest relies on charging to enforce some degree of fairness; since OpenDHT is currently deployed in an environment where such charging is both impractical and impolitic, we wanted a way to achieve fairness without an explicit economic incentive.

Our goals for the OpenDHT storage allocation algorithm are as follows. First, to simplify life for its clients, OpenDHT should offer storage with a definite time-to-live (TTL). A client should know exactly when it must re-store its puts in order to keep them stored, so rather than adapting (as in Palimpsest), the client can merely set simple timers or forget its data altogether (if, for instance, the application's need for the data will expire before the data itself).

Second, the allocation of storage across clients should be "fair" without invoking explicit charging. By fair we mean that, upon overload, each client has "equal" access to storage.[2] Moreover, we also mean fair in the work-conserving sense; OpenDHT should allow for full utilization of the storage available (thereby precluding quota-like policies), and should restrict clients *only* when it is overloaded.

Finally, OpenDHT should prevent *starvation* by ensuring a minimal rate at which puts can be accepted at all times. Without such a requirement, the system could allocate all its storage (fairly) for an arbitrarily long TTL, and then reject all storage requests for the duration of that TTL. Such "bursty" availability of storage would present an undue burden on OpenDHT clients.

In Section 4 we present an algorithm that meets the above goals.

The preceding was an overview of our design. We next consider the details of the OpenDHT client interface, and thereafter, the details of storage allocation in OpenDHT.

# 3. INTERFACE

One challenge to providing a shared DHT infrastructure is designing an interface that satisfies the needs of a sufficient variety of applications to justify the shared deployment. OpenDHT addresses this challenge two ways. First, a put/get interface makes writing simple applications easy yet still supports a broad range of storage applications. Second, the use of a client-side library called ReDiR allows more sophisticated interfaces to be built atop the base put/get interface. In this section we discuss the design of these interfaces. Section 5 presents their performance and use.

## 3.1 The put/get API

The OpenDHT put/get interface supports a range of application needs, from storage in the style of the Cooperative File System (CFS) [9] to naming and rendezvous in the style of the Host Identity Protocol (HIP) [21] and instant messaging.

The design goals behind the put/get interface are as follows. First, simple OpenDHT applications should be simple to write. The value of a shared DHT rests in large part on how easy it is to use. OpenDHT can be accessed using either Sun RPC over TCP or XML RPC over HTTP; as such it easy to use from most programming languages and works from behind most firewalls and NATs. A Python program that reads a key and value from the console and puts them into the DHT is only nine lines long; the complementary get program is only eleven.

| Procedure | Functionality |
|---|---|
| *put*$(k,v,H(s),t)$ | Write $(k,v)$ for TTL $t$<br>can be removed with secret $s$ |
| *get*$(k)$ returns $\{(v,H(s),t)\}$ | Read all $v$ stored under $k$<br>returned value(s) unauthenticated |
| *remove*$(k,H(v),s,t)$ | Remove $(k,v)$ put with secret $s$<br>$t >$ than TTL remaining for put |
| *put-immut*$(k,v,t)$ | Write $(k,v)$ for TTL $t$<br>immutable $(k = H(v))$ |
| *get-immut*$(k)$ returns $(v,t)$ | Read $v$ stored under $k$<br>returned value immutable |
| *put-auth*$(k,v,n,t,K_P,\sigma)$ | Write $(k,v)$, expires at $t$<br>public key $K_P$; private key $K_S$<br>can be removed using nonce $n$<br>$\sigma = \{H(k,v,n,t)\}_{K_S}$ |
| *get-auth*$(k,H(K_P))$ returns $\{(v,n,t,\sigma)\}$ | Read $v$ stored under $(k,H(K_P))$<br>returned value authenticated |
| *remove-auth*$(k,H(v),n,t,K_P,\sigma)$ | Remove $(k,v)$ with nonce $n$<br>parameters as for *put-auth* |

**Table 1: The put/get interface. $H(x)$ is the SHA-1 hash of $x$.**

Second, OpenDHT should not restrict key choice. Previous schemes for authentication of values stored in a DHT require a particular relationship between the value and the key under which it is stored (*e.g.*, [9, 14]). Already we know of applications that have key choice requirements that are incompatible with such restrictions; the prefix hash tree (PHT) [25] is one example. It would be unwise to impose similar restrictions on future applications.

Third, OpenDHT should provide authentication for clients that need it. A client may wish to verify that an authorized entity wrote a value under a particular key or to protect its own values from overwriting by other clients. As we describe below, certain attacks cannot be prevented without support for authentication in the DHT. Of course, our simplicity goal demands that authentication be only an option, not a requirement.

The current OpenDHT deployment meets the first two of these design goals (simplicity and key choice) and has some support for the third (authentication). In what follows, we describe the current interface in detail, then describe two planned interfaces that better support authentication. Table 1 summarizes all three interfaces. Throughout, we refer to OpenDHT keys by $k$; these are 160-bit values, often the output of the SHA-1 hash function (denoted by $H$), though applications may assign keys in whatever fashion they choose. Values, denoted $v$, are variable-length, up to a maximum of 1 kB in size. All values are stored for a bounded time period only; a client specifies this period either as a TTL or an expiration time, depending on the interface.

Finally, we note that under all three interfaces, OpenDHT provides only eventual consistency. In the case of network partitions or excessive churn, the system may fail to return values that have been put or continue to return values that have been removed. Imperfect clock synchronization in the DHT may also cause values to expire at some replicas before others, leaving small windows where replicas return different results. While such temporary inconsistencies in theory limit the set of applications that can be built on OpenDHT, they have not been a problem to date.

[2]As in fair queuing, we can of course impose weighted fairness, where some clients receive a larger share of storage than others, for policy or contractual reasons. We do not pursue this idea here, but it would require only minor changes to our allocation mechanism.

### 3.1.1 The Current Interface

A put in OpenDHT is uniquely identified by the triple of a key, a value, and the SHA-1 hash of a client-chosen random secret up to 40 bytes in length. If multiple puts have the same key and/or value, all are stored by the DHT. A put with the same key, value, and secret hash as an existing put refreshes its TTL. A get takes a key and returns all values stored under that key, along with their associated secret hashes and remaining TTLs. An iterator interface is provided in case there are many such values.

To remove a value, a client reveals the secret whose hash was provided in the put. A put with an empty secret hash cannot be removed. OpenDHT stores removes like puts, but a DHT node discards a put $(k, v, H(s))$ for which it has a corresponding remove. To prevent the DHT's replication algorithms from recovering this put when the remove's TTL expires, clients must ensure that the TTL on a remove is longer than the TTL remaining on the corresponding put. Once revealed in a remove, a secret should not be reused in subsequent puts. To allow other clients to remove a put, a client may include the encrypted secret as part of the put's value.

To change a value in the DHT, a client simply removes the old value and puts a new one. In the case where multiple clients perform this operation concurrently, several new values may end up stored in the DHT. In such cases, any client may apply an application-specific conflict resolution procedure to decide which of the new values to remove. So long as this procedure is a total ordering of the possible input values, it does not matter which client performs the removes (or even if they all do); the DHT will store the same value in the end in all cases. This approach is similar to that used by Bayou [24] to achieve eventual consistency.

Since OpenDHT stores all values put under a single key, puts are robust against *squatting*, in that there is no race to put first under a valuable key (*e.g.*, $H$("coca-cola.com")). To allow others to authenticate their puts, clients may digitally sign the values they put into the DHT. In the current OpenDHT interface, however, such values remain vulnerable to a denial-of-service attack we term *drowning*: a malicious client may put a vast number of values under a key, all of which will be stored, and thereby force other clients to retrieve a vast number of such chaff values in the process of retrieving legitimate ones.

### 3.1.2 Planned Interfaces

Although the current put/get interface suffices for the applications built on OpenDHT today, we expect that as the system gains popularity developers will value protection against the drowning attack. Since this attack relies on forcing legitimate clients to sort through chaff values put into the DHT by malicious ones, it can only be thwarted if the DHT can recognize and reject such chaff. The two interfaces below present two different ways for the DHT to perform such access control.

**Immutable puts:** One authenticated interface we plan to add to OpenDHT is the immutable put/get interface used in CFS [9] and Pond [28], for which the DHT only allows puts where $k = H(v)$. Clearly, such puts are robust against squatting and drowning. Immutable puts will not be removable; they will only expire. The main limitation of this model is that it restricts an application's ability to choose keys.

**Signed puts:** The second authenticated interface we plan to add to OpenDHT is one where values put are certified by a particular public key, as used for root blocks in CFS. In these puts, a client employs a public/private key pair, denoted $K_P$ and $K_S$, respectively. We call $H(K_P)$ the *authenticator*.

| Procedure | Functionality |
|---|---|
| *join*(*host*, *id*, *namespace*) | adds (*host*, *id*) to the list of hosts providing functionality of *namespace* |
| *lookup*(*key*, *namespace*) | returns (*host*, *id*) in *namespace* whose *id* most immediately follows *key* |

**Table 2: The lookup interface provided using ReDiR.**

In addition to a key and value, each put includes: a nonce $n$ that can be used to remove the value later; an expiration time $t$ in seconds since the epoch; $K_P$ itself; and $\sigma = \{H(k, v, n, t)\}_{K_S}$, where $\{X\}_{K_S}$ denotes the digital signing of $X$ with $K_S$. OpenDHT checks that the digital signature verifies using $K_P$; if not, the put is rejected. This invariant ensures that the client that sent a put knows $K_S$.

A get for an authenticated put specifies *both* $k$ and $H(K_P)$, and returns only those values stored that match both $k$ and $H(K_P)$. In other words, OpenDHT only returns values signed by the private key matching the public key whose hash is in the get request. Clients may thus protect themselves against the drowning attack by telling the DHT to return only values signed by an entity they trust.

To remove an authenticated put with $(k, v, n)$, a client issues a remove request with $(k, H(v), n)$. As with the current interface, clients must take care that a remove expires after the corresponding put. To re-put a value, a client may use a new nonce $n' \neq n$.

We use expiration times rather than TTLs to prevent expired puts from being replayed by malicious clients. As with the current interface, puts with the same key and authenticator but different values will all be stored by the DHT, and a new put with the same key, authenticator, value, and nonce as an existing put refreshes its TTL.

Authenticated puts in OpenDHT are similar to those used for public-key blocks in CFS [9], for *sfrtags* in SFR [33], for *fileIds* in PAST [14], and for AGUIDs in Pond [28]. Like SFR and PAST, OpenDHT allows multiple data items to be stored using the same public key. Unlike CFS, SFR, and PAST, OpenDHT gives applications total freedom over key choice (a particular requirement in a generic DHT service).

## 3.2 ReDiR

While the put/get interface is simple and useful, it cannot meet the needs of all applications. Another popular DHT interface is *lookup*, which is summarized in Table 2. In this interface, nodes that wish to provide some service—packet forwarding, for example—*join* a DHT dedicated to that service. In joining, each node is associated with an identifier *id* chosen from a *key space*, generally $[0 : 2^{160})$. To find a service node, a client performs a lookup, which takes a key chosen from the identifier space and returns the node whose identifier most immediately follows the key; lookup is thus said to implement the successor relation.

For example, in *i*3 [31], service nodes provide a packet forwarding functionality to clients. Clients create (key, destination) pairs called triggers, where the destination is either another key or an IP address and port. A trigger $(k, d)$ is stored on the service node returned by *lookup* $(k)$, and this service node forwards all packets it receives for key $k$ to $d$. Assuming, for example, that the nodes $A$ through $F$ in Figure 2 are *i*3 forwarding nodes, a trigger with key $B \leq k < C$ would be managed by service node $C$.

The difficulty with lookup for a DHT service is the functionality implemented by those nodes returned by the lookup function. Rather than install application-specific functionality into the service, thereby certainly increasing its complexity and possibly reducing its robustness, we prefer that such functionality be supported outside the DHT, while leveraging the DHT itself to per-

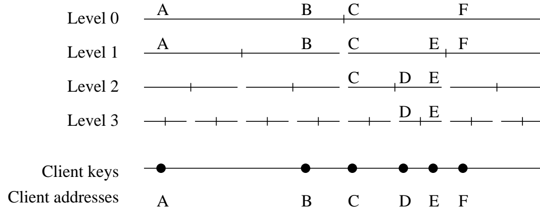


**Figure 2: *An example ReDiR tree with branching factor* $b = 2$. Each tree node is shown as a contiguous line representing the node's interval of the keyspace, and the two intervals associated with each node are separated by a tick. The names of registered application hosts ($A$ through $F$) are shown above the tree nodes at which they would be stored.**

form lookups. OpenDHT accomplishes this separation through the use of a client-side library called ReDiR. (An alternative approach, where application-specific code may only be placed on subsets of nodes *within* the DHT, is described in [18].) By using the ReDiR library, clients can use OpenDHT to route by key among these application-specific nodes. However, because ReDiR interacts with OpenDHT only through the put/get API, the OpenDHT server-side implementation retains the simplicity of the put/get interface.

A DHT supporting multiple separate applications must distinguish them somehow; ReDiR identifies each application by an arbitrary identifier, called its *namespace*. Client nodes providing application-specific functionality join a namespace, and other clients performing lookups do so within a namespace. A ReDiR lookup on identifier $k$ in namespace $n$ returns the node that has joined $n$ whose identifier most immediately follows $k$.

A simple implementation of lookup could be achieved by storing the IP addresses and ports of all nodes that have joined a namespace $n$ under key $n$; lookups could then be performed by getting all the nodes under key $n$ and searching for the successor to the key looked up. This implementation, however, scales linearly in the number of nodes that join. To implement lookup more efficiently, ReDiR builds a two-dimensional quad-tree of the nodes that have joined and embeds it in OpenDHT using the put/get interface.[3] Using this tree, ReDiR performs lookup in a logarithmic number of get operations with high probability, and by estimating the tree's height based on past lookups, it reduces the average lookup to a constant number of gets, assuming uniform-random client IDs.

The details are as follows: each tree node is list of (IP, port) pairs for a subset of the clients that have joined the namespace. An example embedding is shown in Figure 2. Each node in the tree has a *level*, where the root is at level 0, its immediate children are at level 1, *etc.* Given a branching factor of $b$, there are thus at most $b^i$ nodes at level $i$. We label the nodes at any level from left to right, such that a pair $(i, j)$ uniquely identifies the $j$th node from the left at level $i$, and $0 \leq j < b^i$. This tree is then embedded in OpenDHT node by node, by putting the value(s) of node $(i, j)$ at key $H(ns, i, j)$. The root of the tree for the $i3$ application, for example, is stored at $H(\text{“i3”}, 0, 0)$. Finally, we associate with each node $(i, j)$ in the tree $b$ intervals of the DHT keyspace $\left[2^{160} b^{-i}(j + \frac{b'}{b}),\ 2^{160} b^{-i}(j + \frac{b'+1}{b})\right)$ for $0 \leq b' < b$.

We sketch the registration process here. Define $I(\ell, k)$ to be the (unique) interval at level $\ell$ that encloses key $k$. Starting at some level $\ell_{\text{start}}$ that we define later, a client with identifier $v_i$ does an OpenDHT get to obtain the contents of the node associated with $I(\ell_{\text{start}}, v_i)$. If after adding $v_i$ to the list of $(IP, port)$ pairs, $v_i$ is now the numerically lowest or highest among the keys stored in that node, the client continues up the tree towards the root, getting the contents and performing an OpenDHT put in the nodes associated with each interval $I(\ell_{\text{start}} - 1, v_i)$, $I(\ell_{\text{start}} - 2, v_i)$, …, until it reaches either the root (level 0) or a level at which $v_i$ is not the lowest or highest in the interval. It also walks down the tree through the tree nodes associated with the intervals $I(\ell_{\text{start}}, v_i), I(\ell_{\text{start}} + 1, v_i), \ldots$, at each step getting the current contents, and putting its address if $v_i$ is the lowest or highest in the interval. The downward walk ends when it reaches a level in which it is the only client in the interval. Finally, since all state is soft (with TTLs of 60 seconds in our tests), the entire registration process is repeated periodically until the client leaves the system.

A lookup $(ns, k)$ is similar. We again start at some level $\ell = \ell_{\text{start}}$. At each step we get the current interval $I(\ell, k)$ and determine where to look next as follows:

1. If there is no successor of $v_i$ stored in the tree node associated with $I(\ell, k)$, then its successor must occur in a larger range of the keyspace, so we set $\ell \leftarrow \ell - 1$ and repeat, or fail if $\ell = 0$.
2. If $k$ is sandwiched between two client entries in $I(\ell, k)$, then the successor must lie somewhere in $I(\ell, k)$. We set $\ell \leftarrow \ell + 1$ and repeat.
3. Otherwise, there is a client $s$ stored in the node associated with $I(\ell, k)$ whose identifier $v_s$ succeeds $k$, and there are no clients with IDs between $k$ and $v_s$. Thus, $v_s$ must be the successor of $k$, and the lookup is done.

A key point in our design is the choice of starting level $\ell_{\text{start}}$. Initially $\ell_{\text{start}}$ is set to a hard-coded constant (2 in our implementation). Thereafter, for registrations, clients take $\ell_{\text{start}}$ to be the lowest level at which registration last completed. For lookups, clients record the levels at which the last 16 lookups completed and take $\ell_{\text{start}}$ to be the mode of those depths. This technique allows us to adapt to any number of client nodes while usually hitting the correct depth (Case 3 above) on the first try.

We present a performance analysis of ReDiR on PlanetLab in Section 5.2.

# 4. STORAGE ALLOCATION

In Section 2.2, we presented our design goals for the OpenDHT storage allocation algorithm: that it provide storage with a definite time-to-live (TTL), that it allocate that storage fairly between clients and with high utilization, and that it avoid long periods in which no space is available for new storage requests. In this section we describe an algorithm, Fair Space-Time (FST), that meets these design goals. Before doing so, though, we first consider two choices we made while defining the storage allocation problem.

First, in this initial incarnation of OpenDHT, we equate “client” with an IP address (spoofing is prevented by TCP's three-way handshake). This technique is clearly imperfect: clients behind the same NAT or firewall compete with each other for storage, mobile clients can acquire more storage than others, and some clients (*e.g.*, those that own class A address spaces) can acquire virtually unlimited storage. To remedy this situation, we could clearly use a more sophisticated notion of client (person, organization, *etc.*) and require each put to be authenticated at the gateway. However, to be completely secure against the Sybil attack [13], this change would require formal identity allocation policies and mechanisms. In order to make early use of OpenDHT as easy as possible, and to prevent administrative hassles for ourselves, we chose to start with the much more primitive per-IP-address allocation model, and we hope

[3]The implementation of ReDiR we describe here is an improvement on our previous algorithm [19], which used a fixed tree height.

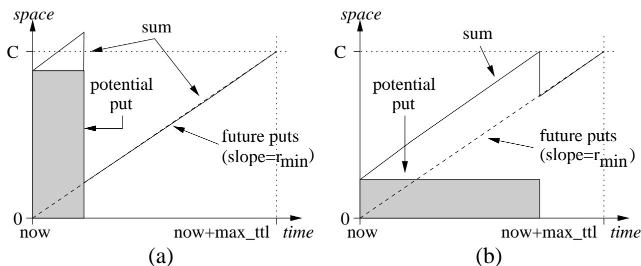


**Figure 3:** ***Preventing starvation.***

to improve on it in the future. More generally, we discuss in Section 6 how our current free service could transition to a competitive commercial market in DHT service.

Second, OpenDHT is a large distributed system, and at first one might think that a fair allocation mechanism should consider the global behavior of every client (*i.e.*, all of their puts). While tracking global behavior in this way presents a daunting problem, it is also the case that the capacity constraints of OpenDHT are per-node, in the form of finite disk capacities, so the situation is even more complicated.[4]

We note that OpenDHT cannot avoid providing some notion of per-disk fairness in allocation. For example, a common use of the system is for rendezvous, where a group of cooperating clients discover each other by putting their identities under a common key, $k$. With a strictly global model of fairness, a malicious client could disrupt this rendezvous by filling the disk onto which $k$ is mapped, so long as it remained below its globally fair allocation. A per-disk model of fairness, in contrast, promises each client a fair allocation of every disk in the system, preventing such attacks.

Furthermore, the per-disk model rewards socially responsible behavior on the part of clients. Applications that are flexible in their key choice—the PAST storage system [14], for example—can target their puts towards otherwise underutilized nodes, thereby balancing the load on the DHT while acquiring more storage for themselves. By protecting applications that cannot choose their keys while rewarding those that can, the per-disk model reduces the need for later load balancing by the DHT itself.

For the above reasons, we have implemented per-disk fairness in OpenDHT, and we leave the study of global fairness to future work. Still, per-disk fairness is not as easy to implement as it sounds. Our storage interface involves both an amount of data (the size of the put in bytes) and a duration (the TTL). As we will see in Section 4, we can use an approach inspired by fair queuing [12] to allocate storage, but the two-dimensional nature of our storage requires substantial extensions beyond the original fair queuing model.

We now turn to describing the algorithmic components of FST. First we describe how to achieve high utilization for storage requests of varied sizes and TTLs while preventing starvation. Next, we introduce the mechanism by which we fairly divide storage between clients. Finally, we present an evaluation of the FST algorithm in simulation.

## 4.1 Preventing Starvation

An OpenDHT node prevents starvation by ensuring a minimal rate at which puts can be accepted at all times. Without such a requirement, OpenDHT could allocate all its storage (fairly) for an arbitrarily large TTL, and then reject all storage requests for the duration of that TTL. To avoid such situations, we first limit all TTLs to be less than $T$ seconds and all puts to be no larger than $B$ bytes. We then require that each OpenDHT node be able to accept at the rate $r_{min} = C/T$, where $C$ is the capacity of the disk. We could choose a less aggressive starvation criterion, one with a smaller $r_{min}$, but we are presenting the most challenging case here. (It is also possible to imagine a reserved rate for future puts that is not constant over time—*e.g.,* we could reserve a higher rate for the near future to accommodate bursts in usage—but as this change would significantly complicate our implementation, we leave it for future work.)

When considering a new put, FST must determine if accepting it will interfere with the node's ability to accept sufficiently many later puts. We illustrate this point with the example in Figure 3, which plots committed disk space versus time. The rate $r_{min}$ reserved for future puts is represented by the dashed line (which has slope $r_{min}$). Consider two submitted puts, a large one (in terms of the number of bytes) with a short TTL in Figure 3(a) and a small one with a long TTL in Figure 3(b). The requirement that these puts not endanger the reserved minimum rate ($r_{min}$) for future puts is graphically equivalent to checking whether the sum of the line $y = r_{min}x$ and the top edge of the puts does not exceed the storage capacity $C$ at any future time. We can see that the large-but-short proposed put violates the condition, whereas the small-but-long proposed put does not.

Given this graphical intuition, we derive a formal admission control test for our allocation scheme. Let $B(t)$ be the number of bytes stored in the system at time $t$, and let $D(t_1,t_2)$ be the number of bytes that free up in the interval $[t_1,t_2)$ due to expiring TTLs. For any point in time, call it $t_{now}$, we can compute as follows the total number of bytes, $f(\tau)$, stored in the system at time $t_{now}+\tau$ assuming that new puts continue to be stored at a minimum rate $r_{min}$:

$$f(\tau) = B(t_{now}) - D(t_{now}, t_{now}+\tau) + r_{min} \times \tau$$

The first two terms represent the currently committed storage that will still be on disk at time $t_{now}+\tau$. The third term is the minimal amount of storage that we want to ensure can be accepted between $t_{now}$ and $t_{now}+\tau$.

Consider a new put with size $x$ and TTL $\ell$ that arrives at time $t_{now}$. The put can be accepted if and only if the following condition holds for all $0 \le \tau \le \ell$:

$$f(\tau) + x \le C. \tag{1}$$

If the put is accepted, the function $f(\tau)$ is updated. Although we omit the details here due to space concerns, this update can be done in time logarithmic in the number of puts accepted by tracking the inflection points of $f(\tau)$ using a balanced tree.

## 4.2 Fair Allocation

The admission control test only prevents starvation. We now address the problem of fair allocation of storage among competing clients. There are two questions we must answer in this regard: how do we measure the resources consumed by a client, and what is the fair allocation granularity?

To answer the first question, we note that a put in OpenDHT has both a size and a TTL; *i.e.*, it consumes not just storage itself, but storage over a given time period. The resource consumed by a put is then naturally measured by the product of its size (in bytes) and its TTL. In other words, for the purposes of fairness in OpenDHT, a put of 1 byte with a TTL of 100 seconds is equivalent to a put of

[4]We assume that DHT load-balancing algorithms operate on longer time scales than bursty storage overloads, so their operation is orthogonal to the concerns we discuss here. Thus, in the ensuing discussion we assume that the key-to-node mapping in the DHT is constant during the allocation process.

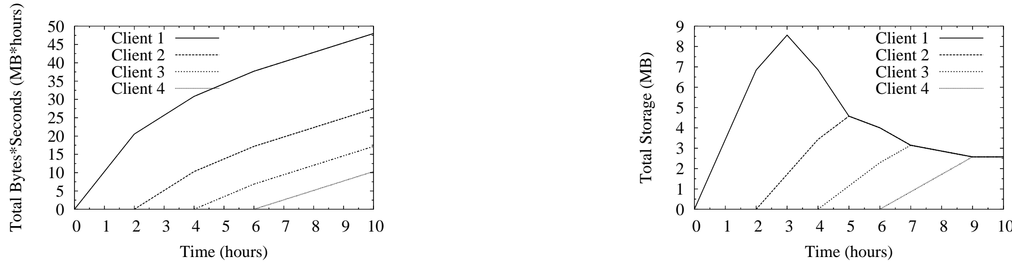


**Figure 4: *Non-starvation.* In this experiment, all clients put above their fair rates, but begin putting at different times.**

100 bytes with a TTL of 1 second. We call the product of the put's size and its TTL its *commitment*.

A straightforward strawman algorithm to achieve fairness would be to track the total commitments made to each client so far, and accept puts from clients with the smallest total commitments. Unfortunately, this policy can lead to per-client starvation. To illustrate this point, assume that client $A$ fills the disk in an otherwise quiescent system. Once the disk is full, client $B$ begins putting its own data. $B$ will not starve, as the admission control test guarantees that the node can still accept data at a rate of at least $r_{min}$, but $A$ will starve because this strawman algorithm favors client $B$ until it reaches the same level of total commitments granted to client $A$. This period of starvation could be as long as the maximum TTL, $T$.

To prevent such per-client starvation, we aim to equalize the *rate* of commitments (instead of the total commitments) of clients that contend for storage. Thus, the service that a client receives depends only on the competing clients at that instant of time, and not on how many commitments it was granted in the past. This strategy emulates the well known *fair queuing* algorithm that aims to provide instantaneous fairness, *i.e.*, allocate a link capacity equally among competing flows at every instant of time.

In fact, our FST algorithm borrows substantially from the start-time fair queuing (SFQ) algorithm [16]. FST maintains a system virtual time $v(t)$ that roughly represents the total commitments that a continuously active client would receive by time $t$. By "continuously active client" we mean a client that contends for storage at every point in time. Let $p_c^i$ denote the $i$-th put of client $c$. Then, like SFQ, FST associates with each put $p_c^i$ a start time $S(p_c^i)$ and a finish time $F(p_c^i)$. The start time of $p_c^i$ is

$$S(p_c^i) = \max(v(A(p_c^i)) - \alpha, F(p_c^{i-1}), 0). \quad (2)$$

$A(p_c^i)$ is the arrival time of $p_c^i$, and $\alpha$ is a non-negative constant described below. The finish time of $p_c^i$ is

$$F(p_c^i) = S(p_c^i) + size(p_c^i) \times ttl(p_c^i).$$

As with the design of any fair queuing algorithm, the key decision in FST is how to compute the system virtual time, $v(t)$. With SFQ the system virtual time is computed as the start time of the packet currently being transmitted (served). Unfortunately, in the case of FST the equivalent concept of *the* put currently being served is not well-defined since there are typically many puts stored in the system at any time $t$. To avoid this problem, FST computes the system virtual time $v(t)$ as the maximum start time of all puts accepted before time $t$.

We now briefly describe how the fairness algorithm works in conjunction with the admission control test. Each node maintains a bounded-size queue for each client with puts currently pending. When a new put arrives, if the client's queue is full, the put is rejected. Otherwise, the node computes its start time and enqueues it. Then the node selects the put with the lowest start time, breaking ties arbitrarily. Using the admission control test (Eqn. 1) the node checks whether it can accept this put right away. If so, the node accepts it and the process is repeated for the put with the next-lowest start time. Otherwise, the node sleeps until it can accept the pending put.

If another put arrives, the node awakes and repeats this computation. If the new put has the smallest start time of all queued puts it will preempt puts that arrived before it. This preemption is particularly important for clients that only put rarely—well below their fair rate. In such cases, the max function in Equation 2 is dominated by the first argument, and the $\alpha$ term allows the client to preempt puts from clients that are at or above their fair rate. This technique is commonly used in fair queuing to provide low latency to low-bandwidth flows [12].

FST can suffer from occasional loss of utilization because of head-of-line blocking in the put queue. However, this blocking can only be of duration $x/r_{min}$, where $x$ is the maximal put size, so the loss of utilization is quite small. In particular, in all of our simulations FST achieved full utilization of the disk.

## 4.3 Evaluation

We evaluate FST according to four metrics: (1) *non-starvation*, (2) *fairness*, (3) *utilization*, and (4) *queuing latency*. We use different maximum TTL values $T$ in our tests, but $r_{min}$ is always 1000 bytes per second. The maximum put size $B$ is 1 kB. The maximum queue size and $\alpha$ are both set to $BT$.

For ease of evaluation and to avoid needlessly stressing PlanetLab, we simulate our algorithm using an event-driven simulator run on a local machine. This simulator tracks relevant features of an OpenDHT node's storage layer, but does not model any network latency or bandwidth. The interval between two puts for each client follows a Gaussian distribution with a standard deviation of 0.1 times the mean. Clients do not retry rejected puts.

Our first experiment shows that FST prevents starvation when clients start putting at different times. In this experiment, the maximum TTL is three hours, giving a disk size of 10.3 MB ($3 \times 3600 \times 1000$ bytes). Each client submits 1000-byte, maximum-TTL puts at a rate of $r_{min}$. The first client starts putting at time zero, and the subsequent clients start putting two hours apart each. The results of the experiment are shown in Figure 4. The left-hand graph shows the cumulative commitments granted to each client, and the right-hand graph shows the storage allocated to each client over time.

Early in the experiment, Client 1 is the only active client, and it quickly acquires new storage. When Client 2 joins two hours later, the two share the available put rate. After three hours, Client 1 continues to have puts accepted (at $0.5r_{min}$), but its existing puts begin to expire, and its on-disk storage decreases. The important point to note here is that Client 1 is not penalized for its past commitments; its puts are still accepted at the same rate as the puts of the Client 2. While Client 1 has to eventually relinquish some of its storage, the

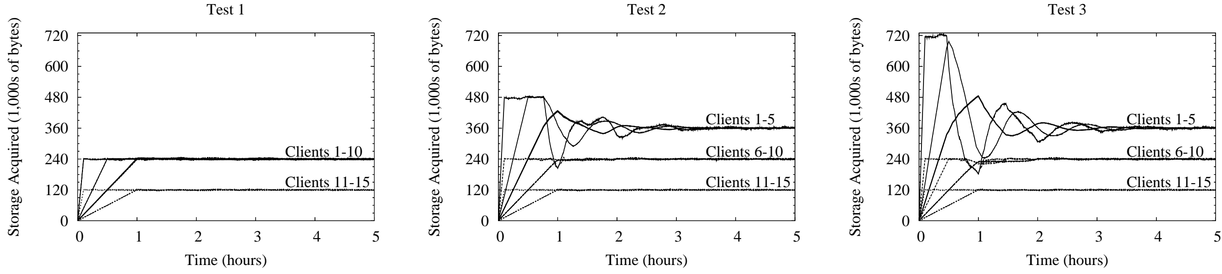


**Figure 5:** ***Fair allocation despite varying put sizes and TTLs.*** **See text for description.**

| Client | Size | TTL | Test 1 | | | | Test 2 | | | | Test 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Bid | 50th | 90th | Avg | Bid | 50th | 90th | Avg | Bid | 50th | 90th | Avg |
| 1 | 1000 | 60 | 1.0 | 0 | 974 | 176 | 2.0 | 5222 | 10851 | 5126 | 3.0 | 6605 | 12949 | 6482 |
| 2 | 1000 | 30 | 1.0 | 0 | 0 | 9 | 2.0 | 7248 | 11554 | 6467 | 3.0 | 7840 | 13561 | 7364 |
| 3 | 1000 | 12 | 1.0 | 0 | 0 | 9 | 2.0 | 8404 | 12061 | 7363 | 3.0 | 8612 | 14173 | 8070 |
| 4 | 500 | 60 | 1.0 | 0 | 409 | 56 | 2.0 | 7267 | 11551 | 6490 | 3.0 | 7750 | 13413 | 7368 |
| 5 | 200 | 60 | 1.0 | 0 | 0 | 13 | 2.0 | 8371 | 12081 | 7349 | 3.0 | 8566 | 14125 | 8035 |
| 6 | 1000 | 60 | 1.0 | 0 | 861 | 163 | 1.0 | 396 | 1494 | 628 | 1.0 | 446 | 2088 | 933 |
| 7 | 1000 | 30 | 1.0 | 0 | 0 | 12 | 1.0 | 237 | 1097 | 561 | 1.0 | 281 | 1641 | 872 |
| 8 | 1000 | 12 | 1.0 | 0 | 0 | 9 | 1.0 | 221 | 1259 | 604 | 1.0 | 249 | 1557 | 940 |
| 9 | 500 | 60 | 1.0 | 0 | 409 | 63 | 1.0 | 123 | 926 | 467 | 1.0 | 187 | 1162 | 770 |
| 10 | 200 | 60 | 1.0 | 0 | 0 | 14 | 1.0 | 0 | 828 | 394 | 1.0 | 6 | 1822 | 804 |
| 11 | 1000 | 60 | 0.5 | 0 | 768 | 160 | 0.5 | 398 | 1182 | 475 | 0.5 | 444 | 1285 | 531 |
| 12 | 1000 | 30 | 0.5 | 0 | 0 | 6 | 0.5 | 234 | 931 | 320 | 0.5 | 261 | 899 | 328 |
| 13 | 1000 | 12 | 0.5 | 0 | 0 | 5 | 0.5 | 214 | 938 | 306 | 0.5 | 235 | 891 | 311 |
| 14 | 500 | 60 | 0.5 | 0 | 288 | 37 | 0.5 | 137 | 771 | 226 | 0.5 | 171 | 825 | 249 |
| 15 | 200 | 60 | 0.5 | 0 | 0 | 7 | 0.5 | 0 | 554 | 103 | 0.5 | 0 | 715 | 131 |

**Table 3:** ***Queuing times in milliseconds for each of the clients in the multiple size and TTL tests.*** **Sizes are in bytes; TTLs are in minutes. A "bid" of 1.0 indicates that a client is putting often enough to fill 1/15th of the disk in an otherwise idle system.**

non-starvation property of the algorithm allows it to intelligently choose which data to let expire and which to renew.

As new clients arrive, the put rate is further subdivided. One maximum TTL after clients stop arriving, each client is allocated its fair share of the storage available on disk.

Our second experiment demonstrates fairness and high utilization when clients issue puts with various sizes and TTLs. In addition, it also shows that clients putting at or below their fair rate experience only slight queuing delays. The maximum TTL in this experiment is one hour, giving a disk capacity of 3.4 MB ($3600 \times 1000$ bytes).

We consider three tests, each consisting of 15 clients divided into three groups, as shown in Table 3. All the clients in a group have the same total demand, but have different put frequencies, put sizes, and TTLs; *e.g.*, a client submitting puts with maximum size and half the maximum TTL puts twice as often as a client in the same group submitting puts with the maximum size and TTL.

The clients in Groups 2 and 3 put at the same rate in each test. The clients in Group 3 are light users. Each of these users demands only $1/30$th of the available storage. For example, Client 11 submits on average a 1000-byte, maximum-TTL put every 30 seconds. As the fair share of each client is $1/15$th of the disk, the puts of the clients from Group 3 should be always accepted. The clients in Group 2 are moderate users, putting at exactly their fair share. For example, Client 6 submits on average one 1000-byte, maximum-TTL put every 15 seconds.

The clients in Group 1 put at a different rate in each test. In Test 1, they put as the same rate as the clients in Group 2. Since clients in Groups 1 and 2 put at their fair share while the clients in Group 3 put below their fair share, the system is underutilized in this test. In Tests 2 and 3, the clients of Group 1 put at twice and three times their fair rate, respectively. Thus, in both these tests the system is overutilized.

Figure 5 and Table 3 summarize the results for this experiment. Figure 5 shows the storage allocated to each client versus time. As expected, in the long term, every client receives its fair share of storage. Moreover, clients that submit puts with short TTLs acquire storage more quickly than other clients when the disk is not full yet. This effect is illustrated by the steep slopes of the lines representing the allocations of some clients at the beginning of each test. This behavior demonstrates the benefit of using the admission control test to rate-limit new put requests: looking back at Figure 3, one can see that many puts with short TTLs can be accepted in a mostly-empty disk without pushing the value of $f(\tau)$ over $C$.

Table 3 shows the queuing delays experienced by each client. This delay is the time a put waits from the moment it arrives at the node until it is accepted. There are three points worth noting. First, as long as the system is underutilized every client experiences very low queuing delays. This point is illustrated by Test 1.

Second, even when the system is overutilized, the clients that issue puts at below or at their fair rate experience low queuing delays. For example, the clients in Group 3 (*i.e.*, Clients 11-15) which issue puts below their fair rate experience average queuing delays of at most 531 ms, while the clients in Group 2 (*i.e.*, Clients 6-10) which issue puts at their fair rate experience average queuing delays no larger than 1 second. One reason clients in Group 3 experience lower queuing delays than clients in Group 2 is the use of parameter $\alpha$ in the computation of the start times (Eqn. 2). Since clients in Group 3 have fewer puts stored than those in Group 2, there are simply more cases when the start times of puts of clients in Group 3

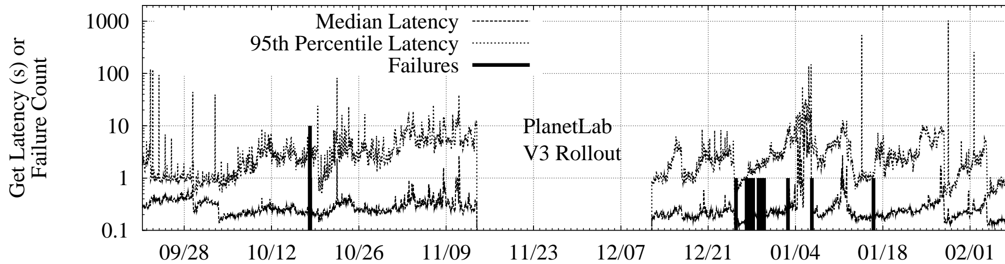


**Figure 6: *Long-running performance and availability of OpenDHT*. See text for description.**

are computed based on the system virtual time (*i.e.*, $v(\cdot) - \alpha$) rather than on the finish times of the previous puts.

Third, clients that are above the fair rate must wait their turn more often, and thus experience higher, but not unreasonable, queuing delays.

# 5. DEPLOYMENT AND EVALUATION

In this section we evaluate both the performance and the usability of OpenDHT.

Much of OpenDHT's routing and storage layers builds on prior efforts. We use the Bamboo DHT implementation for our routing layer [29] and implement a soft-state storage layer atop it similar to that described in Cates' thesis [8]. As such, in evaluating OpenDHT's performance in Section 5.1, we do not focus on the detailed behavior of the underlying DHT routing or storage algorithms, both of which have been evaluated over short periods elsewhere [8, 10, 29]. Rather, we focus on the *long-running* performance of OpenDHT in terms of data durability and put/get latency. Although DHTs are theoretically capable of achieving high durability, we are aware of no previous long term studies of real (not simulated) deployments that have demonstrated this capability in practice.

As discussed in Section 3.2, the ReDiR library presents applications with a lookup interface. Since each ReDiR lookup is implemented using at least one get operation, a lookup in ReDiR can be no faster than a get in the underlying DHT. We quantify the performance of ReDiR lookups on PlanetLab in Section 5.2. This *in situ* performance evaluation is both novel (no implementation of ReDiR was offered or evaluated in [19]) and essential, as the validity of our claim that OpenDHT can efficiently support operations beyond put/get rests largely on the performance penalty of ReDiR *vs.* standard lookup and routing interfaces.

Finally, OpenDHT's usability is best demonstrated by the spectrum of applications it supports, and we describe our early experience with these in Section 5.3.

## 5.1 Long-Running Put/Get Performance

In this section we report on the latency of OpenDHT gets and the durability of data stored in OpenDHT.

**Measurement Setup** OpenDHT has been deployed on PlanetLab since April 2004, on between 170 and 250 hosts. From August 2004 until February 2005 we continuously assessed the availability of data in OpenDHT using a synthetic put/get workload.[5] In this workload, a client puts one value into the DHT each second. Value sizes are drawn randomly from $\{32, 64, 128, 256, 512, 1024\}$ bytes, and TTLs are drawn randomly from $\{1 \text{ hour}, 1 \text{ day}, 1 \text{ week}\}$. The same client randomly retrieves these previously put data to assess their availability; each second it randomly selects one value that should not yet have expired and gets it. If the value cannot be retrieved within an hour, a failure is recorded. If the gateway to which the client is connected crashes, it switches to another, resubmitting any operations that were in flight at the time of the crash.

**Results** Figure 6 shows measurements taken over 3.5 months of running the above workload. We plot the median and 95th percentile latency of get operations on the $y$ axis. The black impulses on the graph indicate failures. Overall, OpenDHT maintains very high durability of data; over the 3.5 months shown, the put/get test performed over 9 million puts and gets each, and it detected only 28 lost values. Get latency is good, although there is some room for improvement. Some of our high latency is due to bugs; on February 4 we fixed a bug that was a major source of the latency "ramps" shown in the graph. On April 22 (not shown) we fixed another and have not seen such "ramps" since. Other high latencies are caused by Internet connectivity failures; the three points where the 95th percentile latency exceeds 200 seconds are due to the gateway being partially partitioned from the Internet. For example, on January 28, the PlanetLab all-pairs-pings database [32] shows that the number of nodes that could reach the gateway dropped from 316 to 147 for 20–40 minutes. The frequency of such failures indicates that they pose a challenge DHT designers should be working to solve.

## 5.2 ReDiR Performance

We consider three metrics in evaluating ReDiR performance: (1) latency of lookups, (2) ReDiR's bandwidth consumption, and (3) consistency of lookups when the registered nodes external to OpenDHT churn. The first two quantify the overhead due to building ReDiR over a put/get interface, while consistency measures ReDiR's ability to maintain correctness despite its additional level of indirection relative to DHTs such as Chord or Bamboo.

**Measurement Setup** To evaluate ReDiR we had 4 PlanetLab nodes each run $n/4$ ReDiR clients for various $n$, with a fifth PlanetLab node performing ReDiR lookups of random keys. We selected an OpenDHT gateway for each set of clients running on a particular PlanetLab node by picking 10 random gateways from a list of all OpenDHT gateways, pinging those ten, and connecting to the one with lowest average RTT. We used a branching factor of $b = 10$ in all of our experiments, with client registration occurring every 30 seconds, and with a TTL of 60 seconds on a client's $(IP, port)$ entries in the tree. Each trial lasted 15 minutes.

**Results** Our first experiment measured performance with a stable set of $n$ clients, for $n \in \{16, 32, 64, 128, 256\}$. Figure 7 shows a

[5]During the PlanetLab Version 3 rollout a kernel bug was introduced that caused a large number of hosts to behave erratically until it was fixed. We were unable to run OpenDHT during this period.

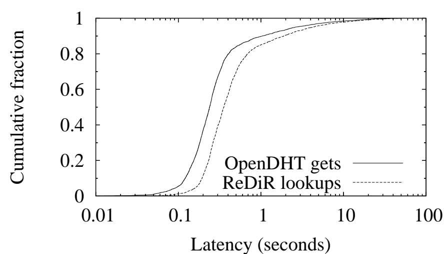


**Figure 7:** ***Latency of ReDiR lookups and OpenDHT gets.***

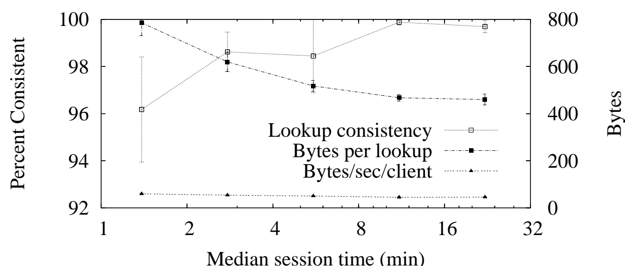


**Figure 8:** ***Percentage of ReDiR lookups that are consistent, bytes transferred per lookup, and bytes/sec per registration process.***

CDF of ReDiR lookup latency, based on 5 trials for each $n$. We compare to the latency of the OpenDHT gets performed in the process of ReDiR lookups. The average lookup uses $\approx 1.3$ gets, indicating that our tree depth estimation heuristic is effective. We have verified this result in a simple simulator for up to 32,768 clients, the results of which match our PlanetLab results closely within their common range of $n$. Bandwidth use is quite low; even at the highest churn rate we tested, the average client registration process uses $\approx 64$ bytes per second and a single lookup uses $\approx 800$ bytes.

We next measured consistency as the rate of client churn varies. We used 128 clients with exponentially distributed lifetimes. Whenever one client died, a new client joined. We use Rhea *et al.*'s definition of consistency [29]: ten lookups were performed simultaneously on the same key, the majority result (if any) is considered consistent, and all others are inconsistent.

Figure 8 plots consistency as a function of median client lifetime. We show the mean and 95% confidence intervals based on 15 trials. Despite its layer of indirection, ReDiR is competitive with the implementation of Chord evaluated in [29], although Bamboo performs better at high churn rates (note, however, that the experiments of [29] were performed on ModelNet, whereas ours were performed on PlanetLab).

In summary, these results show that lookup can be implemented using a DHT service with a small increase in latency, with consistency comparable to other DHTs, and with very low bandwidth.

## 5.3 Applications

We cannot directly quantify the utility of OpenDHT's interface, so in this section we instead report on our experience with building applications over OpenDHT. We first give an overview of the various OpenDHT-based applications built by us and by others. We then describe one application—FreeDB Over OpenDHT (FOOD)—in greater detail. FOOD is a DHT-based implementation of FreeDB, the widely deployed public audio-CD indexing service. As FreeDB is currently supported by a set of replicated servers, studying FOOD allows us to compare the performance of the same application built in two very different ways. We end this section with a brief discussion of common feature requests from application-builders; such requests provide one way to identify which aspects of OpenDHT matter most during development of real applications.

### *5.3.1 Generality: Overview of Applications*

OpenDHT was opened up for experimentation to "friends and family" in March 2004, and to the general public in December 2004. Despite its relative infancy, OpenDHT has already been adopted by a fair number of application developers. To gain experience ourselves, we also developed four different OpenDHT applications. Table 4 lists the known OpenDHT applications. We make a number of observations:

**OpenDHT put/get usage:** Table 4 shows that the majority of these applications use only OpenDHT's put/get interface. We found that many of these (*e.g.*, DOA, FOOD, instant messaging, HIP) make quite trivial use of the DHT—primarily straightforward indexing. Such applications are a perfect example of the benefit of a shared DHT; their relatively simple needs are trivially met by the put/get interface, but none of the applications in themselves warrant the deployment of an independent DHT.

**ReDiR usage:** We have four example applications that use ReDiR—two built by us and two by others. *i*3 is an indirection-based routing infrastructure built over a DHT lookup interface. To validate that ReDiR can be easily used to support applications traditionally built over a lookup interface, we ported the *i*3 code to run over OpenDHT. Doing so was extremely easy, requiring only a simple wrapper that emulated *i*3's Chord implementation and requiring *no* change to how *i*3 itself is engineered.

As described in Section 4, existing DHT-based multicast systems [7, 27, 35] typically use a routing interface. To explore the feasibility of supporting such applications, we implemented and evaluated Multicast Over OpenDHT (MOOD), using a ReDiR-like hierarchy as suggested in [19]. (The QStream project has independently produced another multicast implementation based on a ReDiR-like hierarchy.) MOOD is not a simple port of an existing implementation, but a wholesale redesign. We conjecture based on this experience that one can often redesign routing-based applications to be lookup-based atop a DHT service. We believe this is an area ripe for further research, both in practice and theory.

Finally, the Place Lab project makes novel use of ReDiR. In Place Lab, a collection of independently operated servers processes data samples submitted by a large number of wireless client devices. Place Lab uses ReDiR to "route" an input data sample to the unique server responsible for processing that sample.

In summary, in the few months since being available to the public, OpenDHT has already been used by a healthy number of very different applications. Of course, the true test of OpenDHT's value will lie in the successful, long-term deployment of such applications; we merely offer our early experience as an encouraging indication of OpenDHT's generality and utility.

### *5.3.2 FOOD: FreeDB Over OpenDHT*

FreeDB is a networked database of audio CD metadata used by many CD-player applications. The service indexes over a million CDs, and as of September 2004 was serving over four million read requests per week across ten widely dispersed mirrors.

A traditional FreeDB query proceeds in two stages over HTTP. First, the client computes a hash value for a CD—called its *discid*—and asks the server for a list of CDs with this discid. If only one CD is returned, the client retrieves the metadata for that CD from the server and the query completes. According to our measurements,

| Application | Organization | Uses OpenDHT for ... | put/get or ReDiR | Comments |
|---|---|---|---|---|
| Croquet Media Messenger | Croquet | replica location | put/get | http://opencroquet.org/ |
| Delegation Oriented Arch. (DOA) | MIT, UCB | indexing | put/get | http://nms.lcs.mit.edu/doa/ |
| Host Identity Protocol (HIP) | IETF WG | name resolution | put/get | alternative to DNS-based resolution |
| Instant Messaging Class Project | MIT | rendezvous | put/get | MIT 6.824, Spring 2004 |
| Tetherless Computing | Waterloo | host mobility | put/get | http://mindstream.watsmore.net/ |
| Photoshare | Jordan Middle School | HTTP redirection | put/get | http://ezshare.org/ |
| Place Lab 802.11 Location System | IRS | location-based redirection and range queries | ReDiR | http://placelab.org/ |
| QStream: Video Streaming | UBC | multicast tree construction | ReDiR | http://qstream.org/ |
| RSSDHT: RSS Aggregation | SFSU | multicast tree construction | ReDiR | http://sourceforge.net/projects/rssdht/ |
| FOOD: FreeDB Over OpenDHT | OpenDHT | storage | put/get | 78 semicolons Perl |
| Instant Messaging Over OpenDHT | OpenDHT | rendezvous | put/get | 123 semicolons C++ |
| *i*3 Over OpenDHT | OpenDHT | redirection | ReDiR | 201 semicolons Java glue between *i*3 and ReDiR, passes *i*3 regr. tests, http://i3.cs.berkeley.edu/ |
| MOOD: Multicast Over OpenDHT | OpenDHT | multicast tree construction | ReDiR | 474 semicolons Java |

**Table 4:** ***Applications built or under development on OpenDHT.***

this situation occurs 91% of the time. In the remaining cases, the client retrieves the metadata for each CD in the list serially until it finds an appropriate match.

A single FOOD client puts each CD's data under its discid. To query FOOD, other clients simply get all values under a discid. A proxy that translates legacy FreeDB queries to FOOD queries is only 78 semicolons of Perl.

**Measurement Setup** We stored a May 1, 2004 snapshot of the FreeDB database containing a total of 1.3 million discids in OpenDHT. To compare the availability of data and the latency of queries in FreeDB and FOOD, we queried both systems for a random CD every 5 seconds. Our FreeDB measurements span October 2–13, 2004, and our FOOD measurements span October 5–13.

**Results** During the measurement interval, FOOD offered availability superior to that of FreeDB. Only one request out of 27,255 requests to FOOD failed, where each request was tried exactly once, with a one-hour timeout. This fraction represents a 99.99% success rate, as compared with a 99.9% success rate for the most reliable FreeDB mirror, and 98.8% for the least reliable one.

In our experiment, we measured both the total latency of FreeDB queries and the latency of only the *first* HTTP request within each FreeDB query. We present this last measure as the response time FreeDB might achieve via a more optimized protocol. We consider FreeDB latencies only for the most proximal server, the USA mirror. Comparing the full legacy version of FreeDB against FOOD, we observe that over 70% of queries complete with lower latency on FOOD than on FreeDB, and that for the next longest 8% of queries, FOOD and FreeDB offer comparable response time. For the next 20% of queries, FOOD has less than a factor of two longer-latency than FreeDB. Only for the slowest 2% of queries does FOOD offer significantly greater latency than FreeDB. We attribute this longer tail to the number of request/response pairs in a FOOD transaction *vs.* in a FreeDB transaction. Even for the idealized version of FreeDB, in which queries complete in a single HTTP GET, we observe that roughly 38% of queries complete with lower latency on FOOD than on the idealized FreeDB, and that the median 330 ms required for FOOD to retrieve all CDs' data for a discid is only moderately longer than the median 248 ms required to complete only the first step of a FreeDB lookup.

In summary, FOOD offers improved availability, with minimal development or deployment effort, and reduced latency for the majority of queries *vs.* legacy FreeDB.

### 5.3.3 Common Feature Requests

We now briefly report experience we have gleaned in interactions with users of OpenDHT. In particular, user feature requests are one way of identifying which aspects of the design of a shared DHT service matter most during development of real applications. Requests from our users included:

**XML RPC** We were surprised at the number of users who requested that OpenDHT gateways accept requests over XML RPC (rather than our initial offering, Sun RPC). This request in a sense relates to generality; simple client applications are often written in scripting languages that manipulate text more conveniently than binary data structures, *e.g.*, as is the case in Perl or Python. We have since added an XML RPC interface to OpenDHT.

**Remove function** After XML RPC, the ability to remove values before their TTLs expire was the most commonly requested feature in our early deployment. It was for this reason that we added remove to the current OpenDHT interface.

**Authentication** While OpenDHT does not currently support the immutable or signed puts we proposed in Section 3.1, we have had essentially no requests for such authentication from users. However, we believe this apparent lack of concern for security is most likely due to these applications being themselves in the relatively early stages of deployment.

**Read-modify-write** As discussed in Section 3.1, OpenDHT currently provides only eventual consistency. While it is possible to change values in OpenDHT by removing the old value and putting a new one, such operations can lead to periods of inconsistency. In particular, when two clients change a value simultaneously, OpenDHT may end up storing both new values. Although this situation can be fixed after the fact using application-specific conflict resolution as in Bayou [24], an alternate approach would be to add a read-modify-write primitive to OpenDHT. There has recently been some work in adding such primitives to DHTs using consensus algorithms [22], and we are currently investigating other primitives for improving the consistency provided by OpenDHT.

**Larger maximum value size** Purely as a matter of convenience, several users have requested that OpenDHT support values larger than 1 kB. OpenDHT's current 1 kB limit on values exists only due to Bamboo's use of UDP as a transport. In the near future, we plan to implement fragmentation and reassembly of data blocks in order to raise the maximum value size substantially.

## 6. DISCUSSION

OpenDHT is currently a single infrastructure that provides storage for free. While this is appropriate for a demonstration project, it is clearly not viable for a large-scale and long-term service on which applications critically rely. Thus, we expect that any success trajectory would involve the DHT service becoming a commercial enterprise. This entails two significant changes. First, storage can no longer be free. The direct compensation may not be monetary (*e.g.*, gmail's business model), but the service must somehow become self-sustaining. We don't speculate about the form this charging might take but only note that it will presumably involve authenticating the OpenDHT user. This could be done at the OpenDHT gateways using traditional techniques.

Second, a cooperating but competitive market must emerge, in which various competing DHT service providers (DSPs) peer together to provide a uniform DHT service, a DHT "dialtone," much as IP is a universal dialtone. Applications and clients should be able to access their DSPs (the ones to whom they've paid money or otherwise entered into a contractual relationship) and access data stored by other applications or clients who have different DSPs. We don't discuss this in detail, but a technically feasible and economically plausible peering arrangement is described by Balakrishnan *et al.* [4]. Each DSP would have incentive to share puts and gets with other DSPs, and there are a variety of ways to keep the resulting load manageable. DHT service might be bundled with traditional ISP service (like DNS), so ISPs and DSPs would be identical, but a separate market could evolve.

If such a market emerges, then DHT service might become a natural part of the computational infrastructure on which applications could rely. This may not significantly change the landscape for large-scale, high-demand applications, which could have easily erected a DHT for their own use, but it will foster the development of smaller-scale applications for which the demand is much less certain. Our early experience suggests there are many such applications, but only time will tell.

## 7. SUMMARY

In this paper we have described the design and early deployment of OpenDHT, a public DHT service. Its put/get interface is easy for simple clients to use, and the ReDiR library expands the functionality of this interface so that OpenDHT can support more demanding applications. Storage is allocated fairly according to our per-IP-address and per-disk definition of fairness. The deployment experience with OpenDHT has been favorable; the system is currently supporting a variety of applications, and is slowly building a user community. The latency and availability it provides is adequate and will only get better as basic DHT technology improves.

## 8. ACKNOWLEDGMENTS

We are grateful to Yatin Chawathe, Michael Walfish, and the anonymous reviewers for their excellent feedback, which greatly improved this paper. This work was supported in part under NSF Cooperative Agreement ANI-0225660. Sean Rhea is supported by an IBM Fellowship. Brighten Godfrey is supported by a National Science Foundation Graduate Research Fellowship.

## 9. REFERENCES

[1] Bamboo. `http://bamboo-dht.org/`.
[2] Chord. `http://www.pdos.lcs.mit.edu/chord/`.
[3] Pastry. `http://freepastry.rice.edu/`.
[4] H. Balakrishnan, S. Shenker, and M. Walfish. Peering peer-to-peer providers. In *IPTPS*, Feb. 2005.
[5] A. Bavier et al. Operating system support for planetary-scale network services. In *NSDI*, Mar. 2004.
[6] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable programmable networking. In *FDNA*, 2003.
[7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in a cooperative environment. In *SOSP*, 2003.
[8] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, MIT, May 2003.
[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, Oct. 2001.
[10] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *NSDI*, 2004.
[11] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *IPTPS*, 2003.
[12] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *SIGCOMM*, 1989.
[13] J. Douceur. The Sybil attack. In *IPTPS*, 2002.
[14] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.
[15] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, Mar. 2004.
[16] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *SIGCOMM*, Aug. 1996.
[17] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.
[18] D. R. Karger and M. Ruhl. Diminished Chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *IPTPS*, 2004.
[19] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *IPTPS*, 2004.
[20] A. Mislove et al. POST: a secure, resilient, cooperative messaging system. In *HotOS*, 2003.
[21] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host identity protocol (work in progress). IETF Internet Draft, 2004.
[22] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT-LCS, June 2005.
[23] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, 2002.
[24] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, 1997.
[25] S. Ramabhadran, S. Ratnasamy, J. Hellerstein, and S. Shenker. Brief announcement: Prefix hash tree (extended abstract). In *PODC*, 2004.
[26] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *SIGCOMM*, Aug. 2004.
[27] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. *Lecture Notes in Computer Science*, 2233:14–29, 2001.
[28] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *USENIX FAST*, Mar. 2003.
[29] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Tech. Conf.*, June 2004.
[30] T. Roscoe and S. Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In *HotOS*, May 2003.
[31] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, Aug. 2002.
[32] J. Stribling. Planetlab all-pairs ping. `http://www.pdos.lcs.mit.edu/~strib/pl_app/APP_README.txt`.
[33] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the Web from DNS. In *NSDI*, Mar. 2004.
[34] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JSAC*, 22(1):41–53, Jan. 2004.
[35] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV*, 2001.