Spurring Adoption of DHTs with OpenHash, a Public DHT Service

Brad Karp[†,*] Sylvia Ratnasamy[‡] Sean Rhea[‡,**] Scott Shenker[◇,**]

[†]Intel Research Pittsburgh [*]Carnegie Mellon Univ. [‡]Intel Research Berkeley [**]UC Berkeley [◇]ICSI

1 Introduction

The past three years have seen intense research into Distributed Hash Tables (DHTs): both into algorithms for building them, and into applications built atop them. These applications have spanned a strikingly wide range, including file systems [4, 6, 10], event notification [11], content distribution [2], e-mail delivery [12], indirection services [17, 16], web caches [7], and relational query processors [9]. While this set of applications is impressively diverse, the vast majority of application building is done by a small community of DHT researchers. If DHTs are to have a positive impact on the design of distributed applications *used by real users outside this research community*, we believe that the community of DHT-based application developers should be as broad as possible.

Why, then, has this community of developers remained narrow? First, keeping a research prototype of a DHT running continually requires effort, and experience with DHT code. Second, significant testbed resources are required to deploy and test DHT-based applications. A hacker can download the code for Chord, but she cannot run that code alone; recall that only a tiny fraction of would-be developers has access to a testbed infrastructure like PlanetLab [13]. Consequently, most application developers would turn to ad hoc application-specific solutions rather than attempt to use a DHT.

Our central tenet is that we, as a community, need to harness the ingenuity and talents of the vast majority of application developers who reside outside the rarified but perhaps sterile air of the DHT research community. To that end, we issue a call-to-arms to deploy an *open, publicly accessible* DHT service that would allow new developers to experiment with DHT-based applications *without* the burden of deploying and maintaining a DHT. We believe that there are many simple applications that, individually, might not warrant the effort required to deploy a DHT but that would be trivial to build over a DHT service *were one available*. Many-to-many instant messaging and photo publication, where a user may share photos under a long-lived name even if the photos are served from a home machine with a dynamic IP address are but two of the many such applications. We term these *lite applications*, because they make only simple and often fleeting use of a DHT.[1]

To spur the development of these lite applications we propose *OpenHash*, an open, publicly accessible DHT service that runs on a set of infrastructure hosts and allows any application to execute `put()` and `get()` operations. Its presence would hopefully enable and encourage the development of a wide range of interesting and unexpected DHT applications.[2]

However, we already know from our limited experience with DHT-based applications that some require application-specific processing and thus can't be limited to using the generic `put()/get()` interface. The technical contribution of this paper is an examination of what one can do to extend the functionality of a DHT service so that such application-specific requirements can be reasonably accommodated. That is, we seek to share a common DHT routing platform while allowing application-specific functionality to reside only on certain nodes. To meet this hybrid challenge of shared routing but application-specific processing we propose ReDiR, a distributed rendezvous scheme that removes the need in today's DHT systems to place application-specific code together with DHT code. The ReDiR algorithm itself requires only a `put()/get()` interface from a DHT, but allows a surprisingly wide range of applications to work over a DHT service.

2 DHT as Library *vs.* as Service

Implementing a DHT as a service that exports a narrow `put()/get()` interface is a departure from how present-day DHT-based applications are built. To illuminate the consequences of this design decision, we consider in turn the properties of the current approach and the service approach.

Many existing DHT applications (e.g. [2, 9, 10]) are built using a "bundled" model, where the application is able to read the local DHT state and receive upcalls from the DHT (as in [5]), either by being linked into the same process as

[1]Our claim is not that a DHT service is the *only* or *best* way to build these applications, but rather that a DHT service is a common building block that would be useful for a wide range of such applications, and would be far preferable to building one-off, special-purpose rendezvous or indirection mechanisms for each application (*e.g.,* dynamic DNS for photo publication).

[2]We are often reminded that the most successful peer-to-peer application was developed by a 19-year-old.

the DHT code or through local RPC calls. For brevity, we will say these applications use the DHT as a *library* in either case. To support upcalls, the library model requires that code for the same set of applications be available at all DHT hosts. In practice, this limitation prevents sharing of a single DHT *deployment* by multiple applications—or even by different revisions of the same application—that require distinct upcall-invoked code on all DHT hosts. Instead, different applications only re-use the DHT *code*, amortizing the development of the DHT functionality over the set of applications that use it. A side effect of this is that every such application imposes the maintenance traffic associated with running a DHT on its underlying infrastructure.

The great strength of the library model is the flexibility it affords applications in functionality. By bundling arbitrary application code with DHT code, the model supports any operation at any overlay host on any data held at that host. However, this flexibility comes at the expense of synergy in deployment, and thus at the expense of ease of use of the DHT. We believe that these weaknesses in the library model are the primary reasons for the narrowness of the community of developers of DHT-based applications today. Most would-be DHT-using developers look elsewhere for rendezvous and indirection solutions, because the effort and resources required to deploy a DHT are often greater than those required to hack up a more ad hoc solution.

In contrast, a DHT that provides a more limited interface, specifically only `put()` and `get()`, does not need to be bundled with application code and can thus be deployed as a common, openly accessible service. There are two principal weaknesses of this service model. First, a publicly accessible service is subject to attack.[3] We include a brief discussion in Section 4 but defer a full exploration of the problem to later work. The second problem, which is the focus of this paper, is that the service model is far less flexible in the functionality (and thus the applications) it supports. Intuitively the `put()/get()` interface appears far more restrictive than the "any code, any node" approach of the library model. We offer a detailed discussion of what the service model can and cannot support in the following section. The library model and service model are extremes on a continuum between maximal flexibility without an easily adopted infrastructure, and providing such an infrastructure with reduced flexibility. In the next section, we explore the space between these extremes and the extent to which application-specific functionality can live atop a DHT service.

3 Application-Specific Functionality in OpenHash

Existing, library-based DHT applications [10, 9, 16, 6] co-locate application-specific functionality with the DHT. We distinguish between application functionality invoked at the endpoint (*i.e.*, destination) of a DHT route versus functionality invoked at *every* hop along a route and show that, surprisingly, the former is quite easily achievable over a service while the latter is not.

[3]Note, however, that almost any deployed DHT-based system is subject to attack, whether it uses a DHT library or service.

3.1 Endpoint *vs.* Per-Hop Operators

We use illustrative examples drawn from existing applications, namely PIER and *i3*, to expose the difference between endpoint and per-hop functionality.

The first system we consider is *i3*. In *i3*, packets are *forwarded* to identifiers called triggers. To send a packet to x, a sender routes the packet to the DHT host responsible for storing trigger entries for x. That DHT host then extracts the value I corresponding to x and *forwards* the packet to I. There are two aspects to i3's support for packet forwarding within a DHT. First, each DHT host in i3 must implement the forwarding operation. Second, that forwarding code must read the (key, value) pair stored locally at that DHT host.

Likewise, to execute a join operation in PIER [9], a DHT host iterates over all the (key, value) pairs in its local store, and rehashes them by a portion of their value fields. Here the code for a join operator resides at every DHT host and has full access to the host's local store. The essence of the above behaviors is that the DHT routes an operation request to a key within the keyspace, and the end DHT host responsible for that key carries out an operation that typically accesses the key-value entries stored locally. We term the combination of these behaviors an *endpoint operator.*

A somewhat different example is PIER's computation of aggregates along a tree rooted at a particular key. Nodes route messages toward the root and each DHT host along the path aggregates data before forwarding them. Multicast forwarding within a DHT (as in Scribe, Bayeux, or M-CAN) also uses per-hop processing to set up and maintain dissemination trees. We term such behavior a *per-hop operator*, as it requires application-specific operations be executed at each hop along the path to a key (as opposed to at the final node that holds the key).

3.2 Endpoint Operators in OpenHash

Note that OpenHash cannot by itself support either endpoint or per-hop operators; code for these application-specific operators does not reside at the nodes that constitute OpenHash's DHT, as OpenHash is a shared service. However, we can allow application-specific code to live *outside* of OpenHash and use OpenHash to direct requests to hosts that do support the required operators. This approach allows developers to deploy application-specific code at will while still sharing OpenHash's common key-based routing infrastructure. We term hosts that run the OpenHash DHT *OpenHash hosts* and hosts outside OpenHash that run only application-specific endpoint operators *application hosts*. As with any DHT, application hosts must still divide ownership of their shared keyspace among themselves. In this paper, in the interest of requiring minimal application-specific support *within* OpenHash, we adopt an extreme point in the design space for endpoint operator support, in which application hosts are an entirely disjoint set of hosts from OpenHash hosts. However, the technique we present for supporting endpoint operators works equally well when these sets overlap, or even at the other extreme, where *only* OpenHash hosts implement endpoint operators, and each such host may support different sets of endpoint operators. These two extremes are both important: an application's popularity may warrant its endpoint operators' inclusion in the code at OpenHash hosts, whereas novel endpoint operators for fledgling applications may still be deployed on application hosts without modifying the code installed at OpenHash hosts.

To support endpoint operators, we introduce the notion of *namespaces*. Each application corresponds to a single, uniquely named namespace, and requires a particular set of endpoint operators be available for all keys in its keyspace. OpenHash must route requests destined for key k in application A's namespace (denoted $(A:k)$) to the application host that *both* runs application A *and* is responsible for key k in namespace A. Conventional DHTs consistently hash a set of keys over the hosts that run the DHT itself. We need to solve a different problem: to consistently hash a set of keys over a set of application hosts that may not run the DHT itself. This functionality is effectively the same as that of the `lookup()` interface first proposed in the Chord paper and adopted by the authors in [5] as the KBR or Key-Based Routing interface, with one important difference: our `lookup()` maps application keys to arbitrary application hosts, rather than only to the hosts that run the DHT.

In this section, we describe ReDiR (*Re*cursive *Di*stributed *R*endezvous), a mechanism by which OpenHash can be used to achieve such application-specific `lookup()`s.[4] ReDiR requires hosts that support an endpoint operator for application A to register with OpenHash as application hosts in namespace A. Clients that wish to route to (namespace : key) destinations can then use OpenHash to route to the application host responsible for that key.

ReDiR: There are two interfaces to ReDiR: one for registration of application hosts as members of a namespace, and one for client `lookup()`s of the form `lookup(namespace : key)`, which return the IP address of the application host registered in `namespace` that is responsible for `key`. We describe each in turn, after first describing primitives used by both interfaces.

ReDiR hierarchically decomposes the OpenHash keyspace into nested binary partitions, as shown in Figure 1(a). Level 0 in the decomposition corresponds to the entire keyspace $[0, K-1]$, and generally, level i of the keyspace is decomposed into 2^i partitions, each of length $K/(2^i)$, to a depth of l levels, $i \in [0, l-1]$.[5] Thus, every point in the keyspace has a corresponding set of enclosing partitions, one at every level in the decomposition tree. We assume that all hosts have available the same hash function $H() \rightarrow [0, K-1]$ to map arbitrary data to keys. We define mapping a key $k \in [0, K-1]$ to k' within a narrower subrange of the keyspace $[P, Q]$ by scaling linearly: $k' = P + \lfloor k(Q-P)/K \rfloor$. For a key $k \in [0, K-1]$, when we write "`get(`k`)` or `put(`k`)` *over a subrange of the keyspace* $[P, Q]$," we invoke this mapping implicitly.

Each application that requires endpoint operators has its own namespace, identified by a unique string. Consider an application host X that runs endpoint operator code for application `ABC`. Essentially, host X registers in namespace `ABC` by walking up the ReDiR hierarchy, `put()`ting its own IP address as it goes, until it finds a predecessor. In detail:

- Host X computes $H(X)$. There is exactly one binary partition (keyspace subrange) at each level i that encloses $H(X)$.

- At each level i, starting at the deepest level (2 in the example in Figure 1(a)) and progressing up the tree toward its root (the partition including the entire keyspace), host X first executes `put(`$H($`ABC`$), X$`)` over the subrange of the keyspace in which $H(X)$ falls at that level. Figure 1(a) shows an example of where $H($`ABC`$)$ falls at each level, given a particular $H(X)$. Host X thus appends its IP address to the list of IP addresses (if any) previously `put()` by other hosts within that partition that previously joined namespace `ABC`. Next, host X executes `get(`$H($`ABC`$)$`)` over the same subrange of the keyspace. Host X then searches the one or more returned IP addresses I_j for a *predecessor*—for an I_j such that $H(I_j) < H(X)$.[6] If for any j this condition is satisfied, and a predecessor has been found, host X has completed the registration process. If no predecessor was found, host X continues walking up the tree, and repeats this entire process at the level of the tree next closest to the root. If host X finds no predecessor at levels 1 or greater, the process terminates when it visits level 0 of the tree and stores its IP address there.

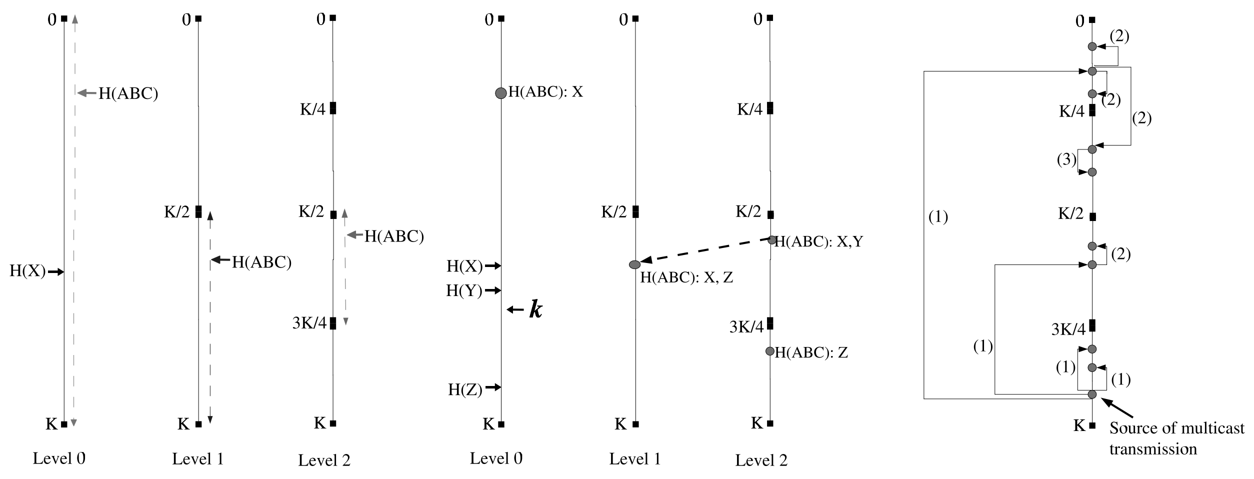
Because application hosts periodically refresh their ReDiR entries, as all `put()`s are timer-expired soft state (see Section 4.1), ReDiR converges to storing at most 2 IP addresses per namespace entry at levels $0 \ldots l-2$ of the hierarchy. At level $l-1$ of the hierarchy, ReDiR stores as many IPs per namespace entry as there are application hosts whose IP addresses fall within that portion of the keyspace; one must choose l commensurately.

A client performs `lookup(ABC :` k`)` as follows:

[4] Note that ReDiR solves a very different problem than bootstrapping multiple application-specific DHTs from a single DHT [3]; we propose a single DHT shared by multiple applications.

[5] In the interest of simplicity, we consider only base-2 decompositions and fixed depth l, though these need not be the same across applications, and could in fact be encoded into the rendezvous name.

[6] There is no modular wrapping here; the comparison is absolute.



(a) ReDiR. (b) Namespaces. (c) Multicast.

Figure 1: *1(a) ReDiR hierarchy and rendezvous points for a node X belonging to application ABC. Dashed lines denote enclosing partitions for X at each level. For clarity, the keyspace at each level is redrawn; in practice there is only one keyspace. 1(b) Using ReDiR to find the correct successor node for application ABC's key k: Node X joined first, then Y and Z. Each rendezvous point lists the nodes registered there. Node Z is key k's correct successor. Dashed lines denote rendezvous nodes contacted to locate k's successor. 1(c) Example of multicast forwarding: transmission links are labeled with their depth in forwarding hierarchy.*

- Note that k falls in one binary partition (keyspace subrange) at each level i of the hierarchy.
- As before, beginning at the deepest level of the tree, and proceeding upward, the client performs a `get(`$H($`ABC`$)$`)` over the keyspace subrange at that level which encloses k. Among the IP addresses I_j returned, if any, the client searches for the successor to k; that is, for the I_j such that $H(I_j)$ is the smallest value satisfying $H(I_j) > k$, where modular wraparound at the high end of the *global* $[0, K-1]$ keyspace is used in the comparison, in the usual fashion when computing DHT keys' successors. When a successor I_j is found at one level of the tree, that I_j is the result of the `lookup()`. If no such successor is found, the search continues up the tree, at the next wider partition enclosing k.

Figure 1(b) shows an example of a client performing `lookup(ABC :` k`)` after the registration process has completed for hosts X, Y, and Z. In the worst case, ReDiR requires a client perform l `get()` operations to complete a `lookup(namespace : key)`; in practice, we expect client-side caching and other optimizations to reduce this cost significantly. We point out that the entire ReDiR mechanism builds exclusively over the simple DHT `put()/get()` interface. To OpenHash, the ReDiR-related (key, value) pairs appear as any others.

Even with ReDiR, OpenHash leaves developers with the burden of deploying application-specific operators. We imagine that over time OpenHash will grow to incorporate some of these more specialized operators, but we don't yet know what this subset should be. Moreover, expecting every node in the OpenHash infrastructure always to run exactly the same set of operators is unrealistically utopian. ReDiR enables a single routing layer to be shared by all services whether partially deployed or not. Without ReDiR, we are stuck with either utopia or bust.

3.3 Per-hop Operators in OpenHash

Although a DHT service with ReDiR performs the route-to-key function for an application, there is no obvious way to support per-hop operators in OpenHash. However, in considering the various forms of per-hop operators described in the literature, we discovered that in most cases one could achieve similar functionality outside of the DHT service, essentially by converting per-hop operations into "scoped" endpoint operations (which ReDiR supports). While we do not claim to know whether all per-hop operators can be implemented using a service, we find this approach promising, and will explore it in greater detail in future work. In this section, we very briefly describe how three sample operations that typically use per-hop operations—multicast, aggregation and server selection (DOLR)—can be built over a DHT service.

Multicast: The following is a brief sketch of one possible solution to multicast in which OpenHash (using ReDiR) provides the rendezvous mechanism, while end nodes implement forwarding. To join group G, a node A inserts (G,A)

within successively largely partitions enclosing $H(A)$ until it hits a partition in which there is already an entry for G. Let d be the maximum depth of the ReDiR hierarchy. Then, to multicast to all of group G, node A does a `get`(G) within its level d partition and unicasts the message to all the group members returned by the `get()`. A also does a `get`(G) within each of its "sibling" partitions from level $d-1$ to 0 and unicasts a message to any *one* node at each level. A node, say B, that receives a message from A assumes that it is responsible for forwarding it on within its half of the smallest partition in the decomposition that contains both A and B; if that decomposition is at level d, B does nothing. Figure 1(c) shows an example of this forwarding. Note however, that unlike schemes like Scribe [2], the above solution need not result in trees optimized for low latency.

Aggregation: Aggregation along the path to a root R could be implemented similarly to multicast by having rendezvous nodes for R at level i aggregate messages before forwarding them on to level $i-1$.

Server selection using DOLR: In systems such as OceanStore and PAST, a client's lookup returns the address of the node closest to the client that stores a copy of the requested object. These systems achieve this through a combination of proximity-sensitive DHT construction and by caching pointers along the path between a node storing a copy and the root node for that object's identifier. This mechanism (often called DOLR) serves two purposes: (1) server selection based on network latency and (2) fate sharing in the sense that if a closeby (*e.g.*, within the client's organization) server is available, the lookup will succeed even if a large fraction of the DHT is unavailable to the client. Both of the above can be achieved without explicitly embedding the supporting functionality into the DHT. For example, [17] and [12] use an org-store and local rings to achieve fate sharing, and [4, 16] use network coordinates to find close copies. In fact, such approaches frequently give applications more control over selection criteria (*e.g.*, server bandwidth or load could be used in place of or in addition to latency). Such flexibility is much harder to achieve using DOLR.

4 Architecture Details

In this section we discuss architectural issues in the design of OpenHash. We begin with the service model, then discuss issues of resource contention.

4.1 Basic Service Model

Figure 2 presents an overview of the OpenHash architecture. Each PlanetLab host runs the OpenHash code and maintains a local store of the (key, value) pairs for which it is responsible. These local stores are accessible to OpenHash clients only through the `put()/get()` interface. Lite applications

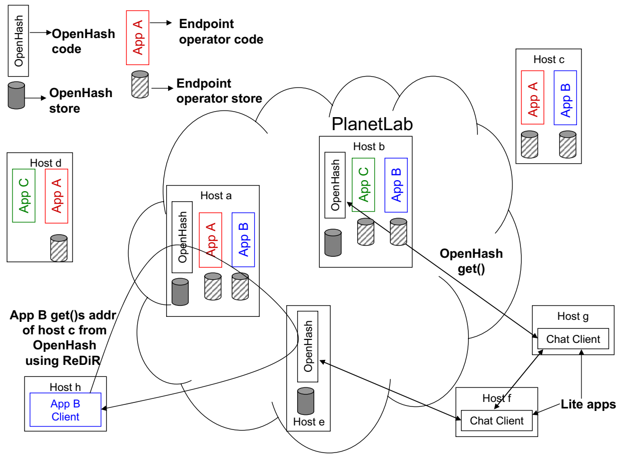


Figure 2: *Overview of the OpenHash architecture. Lite applications, such as instant messaging, use OpenHash only through its* `put()/get()` *interface, while more advanced applications deploy endpoint code and may be co-located with OpenHash servers.*

use only the `put()/get()` interface, and are generally not co-located with OpenHash code and data. In the figure, two instant-messaging clients use OpenHash to discover each other, then communicate over IP. The endpoint operators of more advanced applications may run either by themselves on application hosts, or be co-located with OpenHash on OpenHash hosts. In the figure, endpoint operators for application B run on some of the PlanetLab hosts and on other hosts outside PlanetLab. Endpoint operator instances for B use ReDiR to coordinate amongst themselves. They may also maintain local data stores not managed by OpenHash.

An OpenHash `put()` consists of a client address, key, value, and time to live (TTL). Stored data are soft state; if a value is not refreshed before its TTL expires, OpenHash discards it. Consequently, stored entries do not consume resources indefinitely. `put()`s are uniquely identified by client and key; a subsequent `put()` with the same client and key simply refreshes the TTL, and `put()`s from separate clients do not overwrite each other. Additionally, a client may include a secret with a `put()`; a value thus stored may later be changed by resupplying the secret. We imagine using simple replication (*e.g.*, at k successor nodes) for availability and borrowing from the numerous proposals in the literature for caching and load balancing schemes [10, 14, 16]. An open question is the degree to which OpenHash should expose control over caching and load balancing to its users.

4.2 Resource Contention

A fundamental challenge in providing an open service is that of resource contention; under load, the service should provide each client with a "fair" share of the service's resources. In this section, we briefly put forth three different resource-management techniques as a starting point for discussion.

The first approach is a best-effort service in which OpenHash makes no attempt to arbitrate between the needs of conflicting applications; excess requests are simply discarded and end hosts react to perceived losses by scaling back their rate of resource consumption, perhaps aided by stardardized end-host code as in TCP. This model is easy to implement but vulnerable to faulty or malicious clients that fail to scale back. A more sophisticated approach is to discourage selfish consumption by using fair queueing at every OpenHash node. One challenge with fair queueing is the need for secure identification of clients, though we believe such identification is achievable using SYN-cookie-like techniques [1]. A final approach to resource management is to charge for each use of a resource as has been previously proposed [8, 15], but using computational puzzles instead of micropayments as currency. Unlike with fair queuing, there is no need to securely identify clients in a charge-per-use model.

5 Future Work

Recasting OpenHash as an *active* service, in which DHT nodes download application-specific code, warrants future investigation. For an initial deployment, we believe the simple DHT model is appropriate because it avoids the complexity of downloadable code, yet still supports application-specific operators (with ReDiR), and is, in any case, a prerequisite for an active service.

We view our deployment plans (on PlanetLab, administered by the authors) as largely a bootstrap phase beyond which we imagine OpenHash will evolve to run on infrastructure hosts administered by different authorities. Ensuring robustness when service hosts are not mutually trusting is non-trivial, and the subject of future research.

Acknowledgements

We thank David Tennenhouse, Timothy Roscoe, Hans Mulder, Robert Morris, Frans Kaashoek, Kevin Fall, David Culler, David Mazières, Yatin Chawathe, Michael Walfish, Joe Hellerstein, Petros Maniatis, and John Kubiatowicz for insightful discussions and comments that improved this work.

References

[1] BERNSTEIN, D. Syn cookies. http://cr.yp.to/syncookies.html.

[2] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. SplitStream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of the IPTPS 2003* (2003).

[3] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., AND ROWSTRON, A. One Ring to Rule Them All: Service Discovery and Binding in Structured Peer-to-peer Overlay Networks. In *Proceedings of the 2002 SIGOPS European Workshop* (Sept. 2002).

[4] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)* (Lake Louise, AB, Canada, October 2001).

[5] DABEK, F., ZHAO, B., DRUSCHEL, P., KUBIATOWICZ, J., AND STOICA, I. Towards a Common API for Structured Peer-to-peer Overlays. In *Proceedings of the IPTPS 2003* (Berkeley, February 2003).

[6] DRUSCHEL, P., AND ROWSTRON, A. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)* (Lake Louise, AB, Canada, October 2001).

[7] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing Content Publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004)* (San Francisco, Mar. 2004).

[8] HAND, S., AND ROSCOE, T. Mnemosyne: Peer-to-Peer Steganographic Storage. In *Proceedings of the IPTPS 2002* (Boston, March 2002).

[9] HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. Querying the Internet with PIER. In *Proceedings of VLDB 2003* (Berlin, Germany, September 2003).

[10] KUBIATOWICZ, J. Oceanstore: An Architecture for Global-Scalable Persistent Storage. In *Proceedings of the ASPLOS 2000* (Cambridge, MA, USA, November 2000).

[11] LUIS FELIPE CABRERA, M. B. J., AND THEIMER, M. Herald: Achieving a Global Event Notification Service. In *Proceedings of the HotOS VIII* (May 2001).

[12] MISLOVE, A., POST, A., REIS, C., WILLMANN, P., DRUSCHEL, P., WALLACH, D., BONNAIRE, X., SENS, P., BUSCA, J.-M., AND ARANTES-BEZERRA, L. POST: A Secure, Resilient, Cooperative Messaging System. In *Proceedings of the HotOS IX* (May 2003).

[13] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the HotNets-I 2002* (Princeton, October 2002).

[14] RAO, A., LAKSHMINARAYANAN, K., SURANA, S., KARP, R., AND STOICA, I. Load Balancing in Structured P2P Systems. In *Proceedings of the IPTPS 2003* (Berkeley, February 2003).

[15] ROSCOE, T., AND HAND, S. Palimpsest: Soft-capacity Storage for Planetary-Scale Services. In *Proceedings of the HotOS IX* (May 2003).

[16] STOICA, I., ADKINS, D., ZHUANG, S., SHENKER, S., AND SURANA, S. Internet Indirection Infrastructure. In *Proceedings of the ACM SIGCOMM 2002* (Pittsburgh, PA, USA, August 2002).

[17] WALFISH, M., BALAKRISHNAN, H., AND SHENKER, S. Untangling the Web from DNS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004)* (San Francisco, Mar. 2004).