Towards a Common API for Structured Peer-to-Peer Overlays*

Frank Dabek[1] Ben Zhao[2] Peter Druschel[3] John Kubiatowicz[2] Ion Stoica[2]

[1]MIT Laboratory for Computer Science, Cambridge, MA.
[2]University of California, Berkeley, CA.
[3]Rice University, Houston, TX.

Abstract

In this paper, we describe an ongoing effort to define common APIs for structured peer-to-peer overlays and the key abstractions that can be built on them. In doing so, we hope to facilitate independent innovation in overlay protocols, services, and applications, to allow direct experimental comparisons, and to encourage application development by third parties. We provide a snapshot of our efforts and discuss open problems in an effort to solicit feedback from the research community.

1 Introduction

Structured peer-to-peer overlay networks have recently gained popularity as a platform for the construction or resilient, large-scale distributed systems [6, 7, 8, 10, 11]. Structured overlays conform to a specific graph structure that allows them to locate objects by exchanging $O(\lg N)$ messages where N is the number of nodes in the overlay.

Structured overlays can be used to construct services such as distributed hash tables [4], scalable group multicast/anycast [3, 12], and decentralized object location [5]. These services in turn promise to support novel classes of highly scalable, resilient, distributed applications, including cooperative archival storage, cooperative content distribution and messaging.

Currently, each structured overlay protocol exports a different API and provides services with subtly different semantics. Thus, application designers must understand the intricacies of each protocol and the services they provide to decide which system best meets their needs. Subsequently, applications are locked into one system and unable to leverage innovations in other protocols. Moreover, the semantic differences make a comparative evaluation of different protocol designs difficult.

This work attempts to identify the fundamental abstractions provided by structured overlays and to define APIs for the common services they provide. As the first step, we have identified and defined a *key-based routing API (KBR)*, which represents basic (tier 0) capabilities that are common to all structured overlays. We show that the KBR can be easily implemented by existing overlay protocols and that it allows the efficient implementation of higher level services and a wide range of applications. Thus, the KBR forms the common denominator of services provided by existing structured overlays.

In addition, we have identified a number of higher level (tier 1) abstractions and sketch how they can be built upon the basic KBR. These abstractions include distributed hash tables (DHT), group anycast and multicast (CAST), and decentralized object location and routing (DOLR). Efforts to define common APIs for these services are currently underway.

We believe that defining common abstractions and APIs will accelerate the adoption of structured overlays, facilitate independent innovation in overlay protocols, services, and applications, and permit direct experimental comparisons between systems.

Our APIs will not be universal. Certain applications will wish to use protocol-specific APIs that allow them to exploit particular characteristics of a protocol. This is necessary and desirable to facilitate innovation. However, we expect that such non-standard APIs, once properly understood and abstracted, can be added to the common APIs over time.

The rest of this paper is organized as follows. Section 2 provides an overview of structured overlays and the key services they provide. Next, Section 3 defines and differentiates current tier 1 services. Section 4 describes our KBR API and Section 5 evaluates our proposed API by demonstrating how it can be used to implement a variety of services and how existing overlay protocols can efficiently implement the API. Section 6 discusses future work: developing commons API for higher level tier 1 services like distributed hash tables. We conclude in Section 6.

2 Background

In this section, we define application-visible concepts common to all structured overlay protocols.

A *node* represents an instance of a participant in the overlay (one or more nodes may be hosted by a sin-

*This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660.

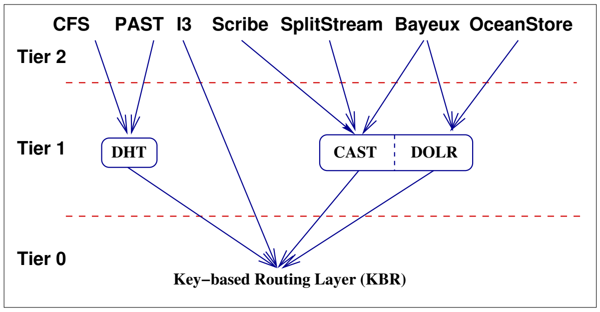

Figure 1: Basic abstractions and APIs, including Tier 1 interfaces: distributed hash tables (DHT), decentralized object location and routing (DOLR), and group anycast and multicast (CAST).

gle physical IP host). Participating nodes are assigned uniform random *nodeIds* from a large *identifier space*. Application-specific objects are assigned unique identifiers called *keys*, selected from the same id space. Tapestry [11, 5], Pastry [8] and Chord [10] use a circular identifier space of n-bit integers modulo 2^n ($n = 160$ for Chord and Tapestry, $n = 128$ for Pastry). CAN [7] uses a d-dimensional cartesian identifier space, with 128-bit nodeIds that define a point in the space.

Each key is dynamically mapped by the overlay to a unique live node, called the key's *root*. To deliver messages efficiently to the root, each node maintains a *routing table* consisting of the nodeIds and IP addresses of the nodes to which the local node maintains overlay links. Messages are forwarded across overlay links to nodes whose nodeIds are progressively closer to the key in the identifier space.

Each system defines a function that maps keys to nodes. In Chord, keys are mapped to the live node with the closest nodeId clockwise from the key. In Pastry, keys are mapped to the live node with the closest nodeId. Tapestry maps a key to the live node whose nodeId has the longest prefix match, where the node with the next higher nodeId value is chosen for each digit that cannot be matched exactly. In CAN, neighboring nodes in the identifier space agree on a partitioning of the space surrounding their nodeIds; keys are mapped to the node responsible for the space that contains the key.

3 Abstractions

All existing systems provide higher level abstractions built upon the basic structured overlays. Examples are Distributed Hash Tables (DHT), Decentralized Object Location and Routing (DOLR), and group anycast/multicast (CAST).

Figure 1 illustrates how these abstractions are related. Key-based routing is the common service provided by all systems at tier 0. At tier 1, we have higher level abstractions provided by some of the existing systems. Most applications and higher-level (tier 2) services use one or more of these abstractions. Some tier 2 systems, like $i3$ [9], use the KBR directly.

The KBR API at tier 0 will be defined in detail in the following section. Here, we briefly explain the tier 1 abstractions and their semantic differences. The key operations of each of these abstractions are sketched in Table 1.

The DHT abstraction provides the same functionality as a traditional hashtable, by storing the mapping between a key and a value. This interface implements a simple store and retrieve functionality, where the value is always stored at the live overlay node(s) to which the key is mapped by the KBR layer. Values can be objects of any type. For example, the DHT implemented as part of the DHash interface in CFS [4] stores and retrieves single disk blocks by their content-hashed keys.

The DOLR abstraction provides a decentralized directory service. Each object replica (or endpoint) has an *objectID* and may be placed anywhere within the system. Applications announce the presence of endpoints by *publishing* their locations. A client message addressed with a particular *objectID* will be delivered to a *nearby* endpoint with this name. Note that the underlying distributed directory can be implemented by annotating trees associated with each *objectID*; other implementations are possible. One might ask why DOLR is not implemented on top of a DHT, with data pointers stored as values; this is not possible because a DOLR routes messages to the nearest available endpoint—providing a locality property not supported by DHTs. An integral part of this process is the maintenance of the distributed directory during changes to the underlying nodes or links.

The CAST abstraction provides scalable group communication and coordination. Overlay nodes may join and leave a group, multicast messages to the group, or anycast a message to a member of the group. Because the group is represented as a tree, membership management is decentralized. Thus, CAST can support large and highly dynamic groups. Moreover, if the overlay that provides the KBR service is proximity-aware, then multicast is efficient and anycast messages are delivered to a group member near the anycast originator.

The DOLR and CAST abstractions are closely related. Both maintain sets of endpoints in a decentralized manner and by their proximity in the network, using a tree consisting of the routes from the endpoints to a common root associated with the set. However, the DOLR abstraction is more tailored towards object location, while the CAST abstraction targets group communication. Thus, their implementations combine different policies with the same basic mechanism. The DHT abstraction, on the other hand, provides a largely orthogonal service, namely a scalable repository for key, value pairs.

DHT	DOLR	CAST
put (key, data)	*publish (objectId)*	*join(groupId)*
remove (key)	*unpublish (objectId)*	*leave(groupId)*
value = get (key)	*sendToObj (msg, objectId, [n])*	*multicast(msg, groupId)* *anycast(msg, groupId)*

Table 1: Tier 1 Interfaces

Defining APIs for the DHT, DOLR and CAST interfaces is the subject of ongoing work. By defining an API for key-based routing and identifying the key tier 1 abstractions, we have taken a major first step.

4 Key-based routing API

In this section we describe the proposed key-based routing API. We begin by defining notation and data types we will use to describe the API. Section 5.1 will show how we can use these calls to implement the DHT, DOLR and CAST higher level abstractions.

4.1 Data types

A *key* is a 160-bit string. A *nodehandle* encapsulates the transport address and nodeId of a node in the system. The nodeId is of type *key*; the transport address might be, for example, an IP address and port. Messages (type *msg*) contain application data of arbitrary length.

We adopt a language-neutral notation for describing the API. A parameter p will be denoted as $\rightarrow p$ if it is a read-only parameter and $\leftrightarrow p$ if it is a read-write parameter. We denote an ordered set *p* of objects of type *T* as *T[] p*.

4.2 Routing messages

void route(key →K, msg →M, nodehandle →hint) This operation forwards a message, *M*, towards the root of key *K*. The optional *hint* argument specifies a node that should be used as a first hop in routing the message. A good hint, e.g. one that refers to the key's current root, can result in the message being delivered in one hop; a bad hint adds at most one extra hop to the route. Either *K* or *hint* may be NULL, but not both. The operation provides a best-effort service: the message may be lost, duplicated, corrupted, or delayed indefinitely.

The **route** operation delivers a message to the key's root. Applications process messages by executing code in upcalls which are invoked by the KBR routing system at nodes along a message's path and at its root. To permit event-driven implementations, upcall handlers must not block and should not perform long-running computations.

void forward(key ↔K, msg ↔M, nodehandle ↔nextHopNode) This upcall is invoked at each node that forwards message *M*, including the source node, and the key's root node (before deliver is invoked). The upcall informs the application that message *M* with key *K* is about to be forwarded to *nextHopNode*. The application may modify the *M*, *K*, or *nextHopNode* parameters or terminate the message by setting *nextHopNode* to NULL.

By modifying the *nextHopNode* argument the application can effectively override the default routing behavior. We will demonstrate examples of the use of this flexibility in Section 5.1.

void deliver(key →K, msg →M) This function is invoked on the the node that is the root for key *K* upon the arrival of message *M*. The **deliver** upcall is provided as a convenience for applications.

4.3 Routing state access

The API allows applications to access a node's routing state via the following calls. All of these operations are strictly local and involve no communication with other nodes. Applications may query the routing state to, for instance, obtain nodes that may be used by the forward upcall above as a next hop destination.

Some of the operations return information about a key's r-root. The r-root is a generalization of a key's root. A node is an r-root for a key if that node becomes the root for the key if all of the i-roots fail for $i < r$. The node may be the r-root for keys in one or more contiguous regions of the ID space.

nodehandle[] local_lookup(key →K, int →num, boolean →safe) This call produces a list of nodes that can be used as next hops on a route towards key *K*, such that the resulting route satisfies the overlay protocol's bounds on the number of hops taken.

If *safe* is true, the expected fraction of faulty nodes in the list is guaranteed to be no higher than the fraction of faulty nodes in the overlay; if false, the set may be chosen to optimize performance at the expense of a potentially higher fraction of faulty nodes. This option allows applications to implement routing in overlays with byzantine node failures. Implementations that assume fail-stop behavior may ignore the *safe* argument. The fraction of faulty nodes in the returned list may be higher if the *safe* parameter is not true because, for instance, malicious nodes have caused the local node to build a routing table that is biased towards malicious nodes [1].

nodehandle [] neighborSet (int →num) This operation produces an unordered list of nodehandles that are neighbors of the local node in the ID space. Up to *num* node handles are returned.

nodehandle [] replicaSet (key →k, int →max_rank) This operation returns an ordered set of nodehandles on which replicas of the object with key *k* can be stored. The call returns nodes with a rank up to and including *max_rank*. If *max_rank* exceeds the implementation's maximum replica set size, then its maximum replica set is returned. Some protocols ([11], [7]) only support a *max_rank* value of one. With protocols that support a rank value greater than one, the returned nodes may be used for replicating data since they are precisely the nodes which become roots for the key *k* when the local node fails.

update(nodehandle →n, bool →joined) This upcall is invoked to inform the application that node n has either joined or left the neighbor set of the local node as that set would be returned by the neighborSet call.

boolean range (nodehandle →N, rank →r, key ↔lkey, key ←rkey) This operation provides information about ranges of keys for which the node N is currently a r-root. The operations returns *false* if the range could not be determined, *true* otherwise. It is an error to query the range of a node not present in the neighbor set as returned by the **update** upcall or the **neighborSet** call. Certain implementations may return an error if r is greater than zero. $[lkey, rkey]$ denotes an inclusive range of key values.

Some protocols may have multiple, disjoint ranges of keys for which a given node is responsible. The parameter *lkey* allows the caller to specify which region should be returned. If the node referenced by N is responsible for key *lkey*, then the resulting range includes *lkey*. Otherwise, the result is the nearest range clockwise from *lkey* for which N is responsible.

5 Validating the API

To evaluate our proposed API, we show how it can be used to implement the tier 1 abstractions, and give examples of other common usages. We believe that the API is expressive enough to implement all the applications known to the authors that have to date been built on top of CAN, Chord, Pastry and Tapestry. We also discuss how the API can be supported on top of several representative structured overlay protocols.

5.1 Use of the API

Here we briefly sketch how tier 1 abstractions (DHT, DOLR, CAST) can be implemented on top of the routing API. We also show how to implement a tier 2 application, Internet Indirection Infrastructure [9], and other mechanisms and protocols such as caching and replication.

DHT. A distributed hash table (DHT) provides two operations: (1) *put(key, value)*, and (2) *value = get(key)*. A simple implementation of *put* routes a *PUT* message containing *value* and the local node's nodehandle, S, using **route(key, [PUT,value,S], NULL)**. The key's root, upon receiving the message, stores the (key, value) pair in its local storage. If the value is large in size, the insertion can be optimized by returning only the nodehandle *R* of the key's root in response to the initial *PUT* message, and then sending the value in a single hop using **route(key, [PUT,value], R)**.

The *get* operation routes a *GET* message using **route(key, [GET,S], NULL)**. The key's root returns the value and its own nodehandle in a single hop using **route(NULL, [value,R], S)**. If the local node remembers *R* from a previous access to *key*, it can provide *R* as a hint.

CAST. Group communication is a powerful building block in many distributed applications. We describe one approach to implementing the CAST abstraction described in Section 3. A key is associated with a group, and the key's root becomes the root of the group's multicast tree. Nodes join the group by routing a SUBSCRIBE message containing their nodehandle to the group's key.

When the **forward** upcall is invoked at a node, the node checks if it is a member of the group. If so, it terminates the SUBSCRIBE message; otherwise, it inserts its nodehandle into the message and forwards the message towards the group key's root, thus implicitly subscribing to the group. In either case, it adds the nodehandle of the joining node to its list of children in the group multicast tree.

Any overlay node may multicast a message to the group, by routing a MCAST message using the group key. The group key's root, upon receiving this message, forwards the message to its children in the group's tree, and so on recursively. To send an anycast message, a node routes an ACAST message using the group key. The first node on the path that is a member of the group forwards the message to one of its children and does not forward it towards the root (returns NULL for nexthop). The message is forwarded down the tree until it reaches a leaf, where it is delivered to the application. If the underlying KBR supports proximity, then the anycast receiver is a group member near the anycast originator.

DOLR. A decentralized object location and routing (DOLR) layer allows applications to control the placement of objects in the overlay. The DOLR layer provides three operations: *publish(objectId)*, *unpublish(ObjectID)*, and *sendToObj(msg, objectId, [n])*.

The *publish* operation announces the availability of an object (at the physical node that issues this operation) under the name *objectID*. The simplest form of *publish* calls **route(objectId, [PUBLISH, objectId, S], NULL)**, where *S* is the name of the originating node. At each hop, an application upcall handler stores a local mapping from *objectId* to *S*. More sophisticated versions of *publish* may deposit pointers along secondary paths to the root. The *unpublish* operation walks through the same path and removes mappings.

The *sendToObj* operation delivers a message to n nearby replicas of a named object. It begins by routing the message towards the object root using **route(objectId, [n, msg], NULL)**. At each hop, the upcall handler searches for local object references matching objectId and sends a copy of the message directly to the closest n locations. If fewer than n pointers are found, the handler decrements n by the number of pointers found and forwards the original message towards objectID by again calling **route(objectId, [n, msg], NULL)**.

Internet Indirection Infrastructure (*i3*). $i3$ is a communication infrastructure that provides indirection, that is, it decouples the act of sending a packet from the act of receiving it [9]. This allows $i3$ to provide support for mobility, multicast, anycast and service composition.

There are two basic operations in $i3$: sources send packets to a logical *identifier* and receivers express interest in packets by inserting a *trigger* into the network. In their simplest form, packets are of the form $(id, data)$ and triggers are of the form $(id, addr)$, where $addr$ is either an identifier or an IP address.[1] Given a packet $(id, data)$, $i3$ will search for a trigger $(id, addr)$ and forward $data$ to $addr$. $i3$ IDs in packets are matched with those in triggers using longest prefix matching. $i3$ IDs are 256-bit long, and their prefix is at least 128-bit long.

To insert a trigger $(id, addr)$, the receiver calls **route(**$H(id_{255:128})$**, [**$id_{127:0}$**, addr], NULL)**, where $H()$ is a hash function that converts an 128-bit string into an unique 160-bit string (eventually by padding $id_{255:128}$ with zeros). This message is routed to the node responsible for $H(id_{255:128})$, where the trigger is stored. Note that all triggers whose IDs have the same prefix are stored at the same node; thus the longest prefix matching is done locally. Similarly, a host sending a packet $(id, data)$ invokes **route(**$H(id_{255:128})$**, [**$id_{127:0}$**, data], NULL)**. When the packet arrives at the node responsible for $H(id_{255:128})$, the packet's id is matched with the trigger's id and forwarded to the corresponding destination. To improve efficiency, a host may cache the address S of the server where a particular id is stored, and use S as a hint when invoking the **route** primitive for that id.

Replication. Applications like DHTs use replication to ensure that stored data survives node failure. To replicate a newly received key (k) r times, the application calls **replicaSet (k,r)** and sends a copy of the key to each returned node. If the implementation is not able to return r suitable neighbors, then the application itself is responsible for determining replica locations.

[1] To support service composition and scalable multicast, $i3$ generalizes the packet and trigger formats by replacing the id of a packet and the $addr$ field of a trigger with a stack of identifiers. However, since these generalizations do not affect our discussion, we ignore them here.

Data Maintenance. When a node's identifier neighborhood changes, the node will be required to move keys to preserve the mapping of keys to nodes, or to maintain a desired replication level. When the **update** upcall indicates that node (n) has joined the identifier neighborhood, the application calls **range (n, i)** with $i = 0 \ldots r$ and transfers any keys which fall in the returned range to n. This has the effect of both transferring data to a node which has taken over the local node's key space ($i = 0$) and maintaining replicas ($i > 0$). This description assumes that a node is using r replicas as returned by **replicaSet**.

Caching. Applications like DHTs use dynamic caching to create transient copies of frequently requested data in order to balance query load. It is desirable to cache data on nodes that appear on the route request messages take towards a key's root because such nodes are likely to receive future request messages. A simple scheme places a cached a copy of a data item on the last node of the route prior to the node that provides the data. Caching can be implemented as follows. A field is added to the request message to store the nodehandle of the previous node on the path. When the **forward** upcall is invoked, each node along the message's path checks whether it stores the requested data. If not, it inserts its nodehandle into the message, and allows the lookup to proceed. If the node does store the data, it sends the data to the requester and sends a copy of the data to the previous node on the request path. The node then terminates the request message by setting *nextHopNode* to NULL.

5.2 Implementation

Here we sketch how existing structured overlay protocols can implement the proposed API. While the chosen example systems (CAN, Chord, Pastry, Tapestry) do not constitute an exhaustive list of structured overlays, they represent a cross-section of existing systems and support our claim the the API can be widely implemented easily.

5.2.1 CAN

The **route** operation is supported by existing operations, and the hint functionality can be easily added. The **range** call returns the range associated with the local node, which in CAN can be represented by a binary prefix. **local_lookup** is a local routing table lookup and currently ignores the value of **safe**. The **update** operation is triggered every time a node splits its namespace range, or joins its range with that of a neighbor.

5.2.2 Chord

Route is implemented in an iterative fashion in Chord. At each hop, the local node invokes an RPC at the next node in the lookup path; this RPC invokes the appropriate upcall (route or deliver) and returns a next hop node. If a hint is given, it is used as the first hop in the search instead of a node taken from the local routing table. The **local_lookup** call returns the closest num successors of K in the node's location table. Calls to **neighborSet** and **replicaSet** return the node's successor list; **neighborSet** calls additionally return the node's predecessor. The range call can be implemented by querying the successor list; given the nth node, it returns the range $[succ[n].ID, succ[n+1].ID$. The exception to this rule is the predecessor; the range of the predecessor cannot be determined.

5.2.3 *Pastry*

The **route** operation can be trivially implemented on top of Pastry's route operation. The hint argument, if present, supersedes the routing table lookup. The **range** operation is implemented based on nodeId comparisons among the members of Pastry's leafset. **local_lookup** translates into a simple lookup of Pastry's routing table; if *safe* is true, the lookup is performed in Pastry's *constrained* routing table [1]. The **update** operation is triggered by a change in Pastry's leafset, and the neighbor set (returned by **neighborSet**) consists of the leafset.

5.2.4 *Tapestry*

The **route** operation is identical to the Tapestry API call *TapestryRouteMsg* forwarded to the hint argument, if present. Tapestry routing tables optimize performance and maintain a small set (generally three) of nodes which are the closest nodes maintaining the next hop prefix matching property. The **local_lookup** call retrieves the optimized next hop nodes. The *safe* routing mode is not used by the current Tapestry implementation, but may be used in future implementations. The **range** operation returns a set of ranges, one each for all combinations of levels where the node can be surrogate routed to. The **update** operation is trigged when a node receives an acknowledged multicast for a new inserting node, or when it receives an object movement request during node deletion [5].

6 Discussion and future work

Settling on a particular key-based routing API were complicated by the tight coupling between applications and the lookup systems on which they were developed. Current block replication schemes, especially the neighbor set replication used by Chord and Pastry, are closely tied to the manner in keys are mapped to nodes. Supporting efficient data replication independent of the lookup system necessitates the **range** and **replicaSet** calls which allow a node to determine where to replicate keys. The common practice of caching blocks along probable lookup paths also requires additional flexibility in the API, namely the upcall mechanism which allows application procedures to execute during the lookup.

The KBR API described here is intended to be language neutral to allow the greatest possible flexibility for implementors of lookup systems. Without specifying a precise binding of the API in a language, application developers will not be able to trivially change which system they use. Instead, the API directs developers to structure their applications in such a way that they can be translated from one system to another with a minimum of effort. One possibility for true portability among structured P2P systems would be to implement the API as an RPC program.

In the future, we will better articulate APIs for tier 1 services such as DHT, DOLR and CAST, including clear definitions of functional and performance expectations. We made a stab at this in Section 3, but more work must be done. In particular, the similarities between DOLR and CAST are striking and demand further exploration. It is at level of tier 1 abstractions that structured peer-to-peer overlays take on their greatest power and utility. We hope that the effort detailed in this paper is the beginning of convergence of functionality toward common services available for all peer-to-peer applications writers.

References

[1] CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. S. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of OSDI* (December 2002).

[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 (IPTPS'03)* (February 2003).

[3] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., AND ROWSTRON, A. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC 20*, 8 (Oct. 2002).

[4] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of SOSP* (Oct. 2001).

[5] HILDRUM, K., KUBIATOWICZ, J. D., RAO, S., AND ZHAO, B. Y. Distributed object location in a dynamic network. In *Proceedings of SPAA* (Winnipeg, Canada, August 2002), ACM.

[6] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of (IPTPS)* (2002).

[7] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, 2001).

[8] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware* (Nov. 2001).

[9] STOICA, I., ADKINS, D., ZHUANG, S., SHENKER, S., AND SURANA, S. Internet indirection infrastructure. In *Proceedings of SIGCOMM* (August 2002), ACM.

[10] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM* (San Diego, 2001).

[11] ZHAO, B., KUBIATOWICZ, J., AND JOSEPH, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr. 2001.

[12] ZHUANG, S. Q., ZHAO, B. Y., JOSEPH, A. D., KATZ, R. H., AND KUBIATOWICZ, J. D. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of NOSSDAV* (June 2001).