# Overview of Structured Peer-to-Peer Overlay Algorithms

**Chapter** · January 2010

DOI: 10.1007/978-0-387-09751-0_9

# Overview of Structured Peer-to-Peer Overlay Algorithms

Krishna Dhara[1], Yang Guo[2], Mario Kolberg[3], Xiaotao Wu[1]

[1]Avaya Labs Research, USA
[2]Corporate Research, Thomson, USA
[3]University of Stirling, UK

Abstract: This chapter provides an overview of structured Peer-to-Peer overlay algorithms. The chapter introduces basic concepts including geometries, routing algorithms, routing table maintenance, node join/leave behaviour, and bootstrapping of structured Peer-to-Peer overlay algorithms. Based on these key concepts, a number of key overlay algorithms are classified into categories and a brief overview of these algorithms is presented. Finally, the chapter presents an 'on-a-glance' comparison of the presented algorithms and provides an outlook on open research issues.

## 1. Overview

Large scale peer-to-peer systems have been deployed for file, music, and other data sharing applications over the internet. The core of these systems is a peer-to-peer network overlay that could connect millions of users or systems and a network that could dynamically discover data stored at any node. Early versions of such peer-to-peer systems mainly consisted of unstructured overlays that organize nodes into random data structures. These unstructured overlays use techniques such as walking or flooding the nodes in the system for lookup, and are often optimized for some common lookup queries. But, in general, these unstructured overlays are quite unpredictable for finding rare items and for some real-time applications such as voice, video sharing etc.

To overcome these issues, structured overlays are developed to provide deterministic bounds on the data discovery. Structured overlays provide scalable network overlays based on a distributed data structure that supports deterministic behaviour for data lookup. Struc-

tured P2P overlays impose restrictions on node placement in the overlay and hence, improve the efficiency of data lookup. In this chapter we take a closer look at these structured peer-to-peer overlays. Earlier surveys of structured overlays can be found in [1][2][3][4]. Here, we present different geometries and their effect on the performance of structured P2P systems. We categorize structured P2P systems in terms of the bound on numbers of hops required for data lookup and present issues such as node lookup, finger table maintenance, and join/leave properties of the overlays. First we define various terms used in structured P2P systems and present the basic notions of a structured peer-to-peer system. We then introduce various classes of structured overlays and discuss their relative merits in the last section.

Some terms and notions are often used when describing P2P overlay algorithms. The most common ones are described briefly below.

**Structured P2P overlay:** A network overlay that connects nodes using a particular data structure or protocol to ensure that node lookup or data discovery is deterministic.

**Distributed hash table (DHT):** A decentralized or distributed hash table that stores (key, value) pairs and is used for data lookups using a key.

**Key-based routing:** The principle by which a message is routed to the owner of a key k from a node n following the principle that either the node n owns the key or points to a node that is closer to a node that owns k in terms of some key space defined by the DHT.

**Routing table (Finger table):** Data structure, usually a table, at nodes that maintain links to other nodes in the structure.

**Churn:** Rate of node joins and leaves in a peer-to-peer network.

# 2. Basic Features of Structured P2P Overlays/Networks

One way to understand structured P2P overlays/networks and to compare various such systems is to study their defining aspects. These aspects include the geometries or data structures used in overlays, the routing algorithms that are enabled by these data structures, the affects of churn on various geometries, the maintenance of the data structure, and the bootstrapping mechanism. These aspects collectively describe the behaviour of structured P2P overlays. In this section, we present what each of these aspects are and how they impact P2P performance in terms of lookup speed, space consumption, and bandwidth requirement. Notations introduced herein are used in later sections to describe various representative structured P2P overlays.

## 2.1. Geometries

Structured P2P overlays use a number of different geometries to accommodate participating nodes in P2P overlays. The term *geometry* is referred to as a structure to organize nodes in a P2P overlay. The primary goal of these geometries is to enable the deterministic lookup. The cost or performance of lookups in a structured P2P overlay is directly related to how nodes are arranged and how the geometry is maintained when new nodes arrive and when old nodes leave. Further, these geometries have a direct impact on the space requirements and on the churn performance of the P2P overlay.

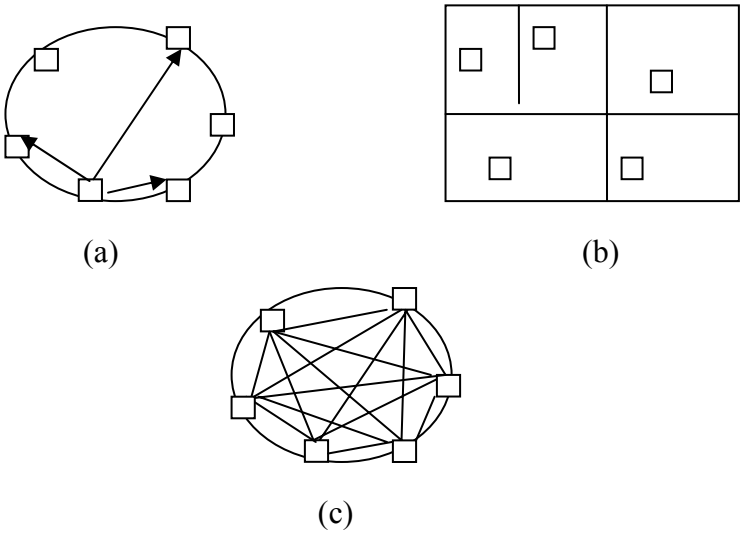


**Figure 1: Examples of Structured P2P Overlay geometries**

Figure 1 shows a few examples of P2P geometries. As depicted in the figure, nodes can be organized in various ways. There are two ways of looking at these structures. One is how the nodes are mapped. In other words how a search of this space proceeds. Another way, though closely related but distinct, is to look at the connectivity of these nodes. For example in Figure 1(a) nodes are organized in a way such that the lookups proceed clock-wise in powers of 2. Each node knows only about a certain number of other nodes in the network. In Figure 1(b) lookups proceed in powers of 2 but in a geometric space. Figure 1(c) shows a node organisation with a high connectivity among nodes. Here the lookups are relatively simple and often only take a single overlay hop.

While structured peer-to-peer overlays offer a uniform distribution of nodes that help in the lookup, the costs associated with the distribution and lookups need to be balanced with the performance under churn, network latency, and space. The scale of peer-to-peer overlays requires significant efforts in maintaining membership changes in overlays. This aspect is further enhanced by the churn of nodes. One way to mitigate this problem is to maintain small tables that do

not require high maintenance. However, small routing tables increase the lookup latency, which is often O(log N)). If the size of routing tables is increased, then structured overlays can reduce the lookup latency as fewer overlay hops are required to reach the destination node. The latency cost can vary between O(log N) to O(1). There are some variations to the O(1) latency overlay algorithms that minimize peer dynamics.

***Logarithmic Overlays:*** Structured overlays use different approaches to route objects. A class of overlays reduce the lookup space by half in each step resulting in a logarithmic number of hops (based on the number of nodes in the overlay). Such overlays are referred to as logarithmic overlays and they guarantee on average O(log N) hops for lookups. Examples of such logarithmic overlays are Chord, Pastry, Tapestry. While Chord uniformly distributes a node across the search space, overlay algorithms like Pastry and Tapestry exploit inter-node proximity while choosing the node's routing table entries. While the average number of hops remains in the same order of complexity with this approach, lower network latency reduces the routing and maintenance costs.

***O(1) Overlays and Constant Overlays:*** In cases where the peer churn is low, the size of the overlay is relatively small, or network latencies for high bandwidth nodes make the routing table maintenance less expensive, constant or O(1) overlays become practical. There are studies that shows that for overlays with millions nodes or more the bandwidth requirements become large and the O(1) overlays become expensive [5] and multi-hop approaches might be preferable. After initial studies on O(log N) overlays, currently there is extensive research on minimizing lookup latency and optimizing the table maintenance costs using constant overlays [6][7].

### 2.2. Routing algorithm

Structured P2P overlays use *routing algorithms* to locate node(s) in an overlay and retrieve data items from them. The routing algorithm defines how a target node is located in the overlay network. This

lookup is closely associated with the geometries of the P2P overlay and the connectivity or information stored at each node.

DHT-based routing algorithms use the hash of a node ID to form a node ID space, which typically is uniformly distributed (however some overlays purposefully break this to achieve a closer relation between the underlying physical network and the overlay). A commonly used hashing function is SHA-1.

The identifier for data items (file name etc) is created by applying the same hashing function. Hence the node IDs and data IDs fall into the same ID space. Data items are typically stored on the closest node with node ID greater than or equal to the data ID. Using this approach each node can find a particular data item using its name. If the node with the closest node ID does not store the data item, it is not available in the network. Using this approach any existing data item can be found by any node in the overlay.

Based on these characteristics a number of different routing algorithms have been defined by various overlays. Major approaches are logarithmic routing (e.g. used by Chord [8]), One-hop routing (EpiChord [9]), XOR routing (e.g. used by Kademlia [10]), and the Content Addressable Network (CAN) [11].

Logarithmic routing means that it takes O (log N) steps to route a message from source to destination node. N is the maximum number of nodes in the overlay. Each node will route a message closer to the destination node by selecting the entry in its routing table whose node ID is closest but smaller or equal to the destination node ID. Logarithmic routing guarantees that with high probability that it does not take more than log N steps to reach a destination node. Overlays described in Section 3 belong to this category.

Clearly, the more accurate and the larger routing tables are, the fewer hops are required to route a message from source to destination. Constant degree overlays guarantee that routing from source to destination is achieved in a certain number of hops, independent of the size of the overlay. Overlays discussed in Section 4 belong to

this category. This approach is pushed to the extreme in one-hop overlay networks which have almost complete routing tables in each node and hence can transmit messages in (almost) a single hop from source to destination. Overlay algorithms described in Section 5 belong to this category.

### 2.3. Join/Leave mechanisms

In the previous sections, we discussed the P2P overlay geometies and the routing mechanisms. P2P systems are highly dynamic in nature. They need robust mechanisms for nodes to join or leave the system at any time with minimal impact to the functioning of the P2P overlay. However, the need for a geometry that leads to a deterministic routing behaviour and the need for autonomous nodes provide a dichotomy for P2P systems. Structured P2P systems use specific join and leave mechanisms for nodes to resolve this dichotomy. These mechanisms provide a balance between high dynamism of P2P systems and a predictable or deterministic P2P overlay behavior.

Peers join an overlay network by connecting themselves to any of the existing peers. But in structured P2P systems, a peer cannot randomly pick exiting peers to join. Instead, it must connect itself to well-defined peers based on its logical identifier and on the geometry of the P2P overlay. Because of this controlled manner, the join and leave mechanisms can greatly affect the performance of structured P2P systems.

***Peer Join:*** Typically, there are three steps for a node to join a structured P2P overlay.

1. The first step is to get a unique identifier for the node. As discussed above, the hashing scheme is based on unique properties of the node. For example, MAC/IP addresses could be used to obtain such an identifier.

2. The second step is to position itself into the overlay structure. This positioning is based on the node's id and the geometry of the P2P overlay. During this step, the node needs to know the entry point or the identity of an existing node to insert itself into the overlay. This process is called *bootstrapping*, which is discussed later in this section.

3. Finally, in the third step, the new joining peer and all the affected peers update their routing tables to stabilize the overall P2P overlay. By *stability* we mean the predictable behaviour of the P2P overlay. This step is often referred to as routing table maintenance.

Routing table maintenance is discussed in the following sections, so we focus on the second step where a node, after finding an existing peer, inserts itself into an overlay. The joining node contacts a peer in the overlay to find out its appropriate position in the overlay. The existing peer uses techniques that are associated with the overlay geometry to find out the new node's neighbours. For example, in Chord, the existing node will issue a lookup to find the successor of the new node based on the new node's identifier. The new peer will then connect to its successor and join the overlay.

Once a new node joins a structured P2P system, the new node and affected peers need to update their properties, usually routing tables, to keep the invariants of the overall system. The number of peers being affected varies for different systems. For example, a new node in Chord affects $O(logN)$ nodes, where $N$ is the number of peers in the system. A new node in CAN affects $O(d)$ nodes, where $d$ is the dimension of the system. However, in O(1) systems, all other nodes in the overlay may need notifications. Clearly, the complexity is dependent on the particular overlay geometry.

***Peer Leave:*** When a node leaves or becomes unreachable in a structured P2P overlay, nodes that point to that node are affected. Their routing table entries will be stale and have to be updated. A timely update results in preserving the invariant of the overlay and guarantees the deterministic lookup in the overlay.

A gracefully departing peer may notify its neighbours about its departure and transfer necessary information to its neighbours for updating their routing tables. Its neighbours then propagate the changes if needed until the invariants of the system are preserved. For example, in Chord, after the update, each node's successor should be correctly maintained and for every key *k*, node *successor (k)* should be responsible for *k*.

In some cases, a node may leave the system unexpectedly, e.g., due to network failure or power outage. Under these circumstances, the node will not notify its neighbours and cannot send necessary information for routing table updates. Hence the system must have a failure detection and stabilization mechanism. Failure detection is usually handled by heartbeat messages or periodic checking. For example, CAN nodes send periodic update messages to their neighbours. The prolonged absence of an update message from a neighbour signals its failure. Chord nodes periodically do random checking on its finger tables to detect failures. Once the failure of a node is detected, usually by its neighbours, the neighbours will start the stabilization process to update routing tables.

## 2.4. Routing table maintenance

As discussed in the previous sections, in structured P2P overlays, each node maintains a routing table to find other peers in the network. Routing table sizes depend heavily on the overlay geometry. In *Multi-hop overlays* (that is a lookup takes multiple overlay hops from source to destination) generally use smaller routing tables than *one-hop overlays*. In one-hop overlays, ideally every node is aware of every other node, hence routing tables need to include references to every other node in the system. This requirement results in large routing tables and poses an additional problem in maintaining the routing tables. Hence, the improved latency behaviour of one-hop overlays comes at a cost of increased maintenance traffic.

Routing table entries need updating if new nodes join or existing nodes leave the network. Usually a join and a graceful leave are propagated through the network by a defined algorithm. However, ungraceful leave events are harder to detect. Generally, two main approaches have been defined for keeping routing tables up-to-date: *opportunistic maintenance* and *active maintenance*.

With opportunistic maintenance, an overlay uses lookup messages and responses to distribute routing table entries. For example, a node will attach entries from its routing table to a response message. The receiver of this response can then augment its routing table with these nodes. This is efficient in terms of number of dedicated maintenance messages required, however, the accuracy of the routing tables is dependent on the number of lookup messages sent. During periods of high churn there is an increased demand for routing table updates. During such periods, opportunistic maintenance may not be sufficient on its own and hence nodes may insert additional lookup messages to receive more routing table updates. As an example, EpiChord [9] discussed in Section 5.3 employs opportunistic routing table maintenance.

With active maintenance, there is a specific algorithm and dedicated messages to propagate routing table entries between nodes. Typically these messages are distributed when a node-join or node-leave event is detected. Usually, neighbouring nodes pick up these events and then distribute these events to all the other nodes in the overlay. Alternatively, update requests are sent after a time interval has expired. Clearly, active maintenance requires a higher bandwidth for distributing node join/leave events, but achieves better routing table accuracy than with opportunistic maintenance schemes. D1HT [46] as discussed in Section 5.4 uses active routing table maintenance.

### 2.5. Bootstrapping

Bootstrapping is a key operation in structured peer-to-peer overlay network. Bootstrapping operation is executed when a peer joins the P2P system for the first time. It enables the initial discovery of other nodes/peers participating in the P2P network. Nascent peers perform

such an operation to join the P2P network. Bootstrapping include the period from peer arrival to the point when the nascent peer becomes a functioning peer of P2P overlay.

One common approach for bootstrapping is through a bootstrapping server. The bootstrapping server maintains a list of participating peers. When contacted by a nascent peer the bootstrapping server returns a partial list of existing peers. The nascent peer connects to the peers in the returned list to join the P2P network. The address of the bootstrap server is usually obtained out-of-band.

Another common approach for bootstrapping is to let nascent peers know in advance an entry point into the network. The entry point can be a list of known peers of a P2P overlay, or a list of non-public bootstrapping servers.

Once a nascent peer gets in touch with some existing peers in the network, it starts the joining process. Different P2P networks employ different strategies, and typically, the joining process is closely related to P2P overlay's routing strategy. The bootstrap server can look at the requesting node's hashed identity and can return the list of existing peers so that the joining process could be optimized. The key issue in designing a good bootstrapping strategy is how to support peers to connect into the network quickly.

## 3. Logarithmic Degree Overlays

In this section as well as Sections 4 (Constant Degree Overlays) and 5 (O(1)-hop overlays), we use the key mechanisms described in the previous section to describe various structured overlays. That is, for each overlay we describe its address space along with its geometry, its routing table and lookup algorithm, and its join and leave mechanisms. In terms of bootstrapping, not all overlays specify a particular approach, however, it appears that all overlays require that a new node knows about at least one peer in the overlay. Common approaches to find an overlay node are IP multicast, using a well-known bootstrap node or using the DNS. Here the overlay service is

associated with a DNS domain name. IP addresses of one or more overlay bootstrap nodes are retrieved using the DNS lookup service.

### 3.1. Chord

Chord [8], developed by a group of researchers at MIT, is one of the first P2P overlay system based on distributed hashing table (DHT).

**Address space:** Chord uses the so-called *consistent hashing* to assign each node and each key an *m*-bit identifier (Id), where *m* is a pre-defined system parameter. Ids fall into the range from 0 to $2^m-1$. Nodes are ordered on an identifier circle modulo $2^m$, as shown in Figure 2. A key is cached at its successor node, defined to be the next node in the identifier circle in the clockwise direction. The predecessor node to a node or key is the next node in the identifier circle in the counter-clockwise direction.

**Routing table and key lookup:** The routing table of Chord nodes consists of two parts. The first part includes a finger table with *m* entries, and the predecessor of this node. Assume the node Id is *n*. The *i*-th entry in the finger table, where $0 \le i < m$, points to the node whose Id is the closest to $n+2^i-1$ in the clock wise direction at identifier circle. Notice that the first entry in the finger table is the successor node of the current node. Predecessor node plus the finger table guarantees the correctness of key lookup service, as described below.

The second part of the routing table is a successor list of size *r*. In addition to the immediate successor node maintained in the finger table, other closest ($r-1$) success nodes are also recorded. The successor list improves the robustness of Chord protocol, and allow Chord to perform correctly in the face of peer churn, i.e., dynamic peer arrivals and departures.

A key lookup request is routed along the identifier. Upon receiving a lookup request, the node first checks if the lookup key Id falls between this node's Id and its successor's Id. If it does then it, returns the successor node as the destination node and terminates the lookup

service. On the other hand, if the lookup key Id does not belong to the current node, the node relays the lookup request to the node in its finger table with Id closest to, but preceding, the lookup key Id. The relaying process proceeds recursively (or iteratively) until the destination node is found. A key lookup example is depicted in Figure 2. In the figure, the left-hand side shows the finger table of Node 8 (N8). Node 16 (N16) appears in four entries in the finger table, while Node 32 (N32) and Node 43 (N43) are also in the finger table. The right-hand side figure depicts the stages for the lookup of key 53 starting from Node 8. It has been shown that the number of routing steps is at the order of *O(log N),* where *N* is the total number of Chord nodes. Refer to [8] for more detailed treatment on Chord routing algorithm.

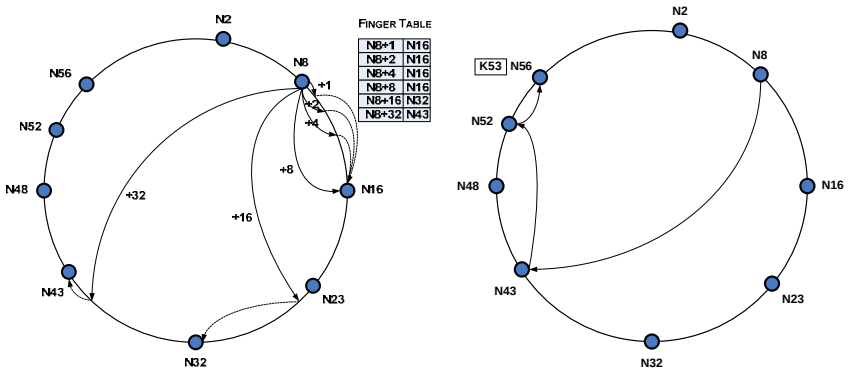


**Figure 2: Example Scenario for key lookup in Chord.**

**Node join and leave:** The newly arrived node in Chord first uses consistent hashing to generate its Id. It then contacts the bootstrapping node, the node already in the Chord, to lookup the successor of its Id. This successor node becomes new node's successor node. The new node uses the stabilization protocol, which is described below, to have a fully correct routing table.

Stabilization protocol is designed to maintain routing tables' correctness in the face of peer churns. It is executed periodically at the background of individual nodes. The stabilization protocol includes following two major functions:

- *Stabilize( ):* allows nodes to learn about newly joined nodes and to update their successor(s) and predecessor.
- *Fix_fingers( ):* ensures finger tables are current and correct.

Node failure/departure creates another challenge to the Chord protocol. The departure of a node leaves its predecessor node's successor pointer invalid, which could affect the routing correctness. To address this issue, Chord maintains a successor list of size *r*. The successor list can be stabilized using slightly changed stabilization protocol. It's proven that with size $r=\Sigma(\log N)$, where $N$ is the total number of nodes in Chord, the lookup can still succeed with high probability even if every node fails with probability of *1/2*. A study of Chord's behavior under churn can be found in [12].

### 3.2. Pastry

Pastry [13][18][19][20][21] is developed by researches from Microsoft Labs Research, Rice University, Purdue University, and University of Washington. There are several applications built on Pastry for different purposes, such as SCRIBE [14][15][16][17] for group communication/event notification, PAST [22][23] for archival storage, SQUIRREL [24] for co-operative web caching, SplitStream [25][26] for high-bandwidth content distribution, POST [27] for co-operative messaging, and Scrivener [28] for fair sharing of resources. Two implementations of Pastry are available for download: FreePastry [29] from Rice University and SimPastry and VisPastry [30] from Microsoft Research.

**Address space:** Each Pastry node has a unique, 128-bit nodeId. Node IDs are chosen randomly and uniformly. One way of generating nodeIds is by hashing nodes' IP addresses.

**Routing table and key lookup:** Pastry uses prefix matching to route messages. Each Pastry node keeps a routing table with $\lceil \log_{2^b} N \rceil$ rows and $2^b - 1$ columns. The entries in row $n$ share the first $n$ digits with the present node. In addition to the routing table, each node also maintains a leaf set that contains the IP addresses of nodes with *l / 2*

numerically closest larger nodeIds, and *l / 2* nodes with numerically closest smaller nodeIds, relative to the present node's nodeId.

Given a message with its key, the node first checks its leaf set. If there is a node whose nodeId is closest to the key, the message is forwarded directly to the node. If the key is not covered by the leaf set, then the node checks the routing table and the message is forwarded to a node that shares a common prefix with the key by at least one more digit. This way, with $\lceil log_2{}^b N \rceil$ steps, the message can reach its destination node.
Figure 3 shows an example lookup scenario. The left-hand side table shows the routing table and the right-hand diagram shows the route. The node *859fdc* looks up a key *d57b2d*. From its routing table, it gets *d13a14*, which shares one digit common prefix with the key. *d13a14* then checks its routing table and get *d52acd*, which shares two digit common prefix with the key. This step keeps on until the key is covered by the node *d57b0c*.

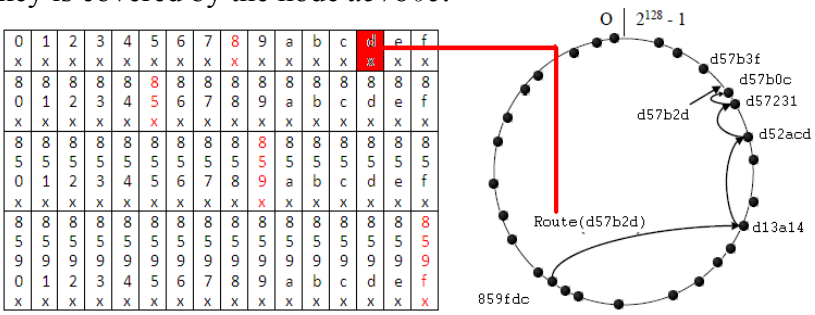


**Figure 3: Example lookup scenario in Pastry.**

**Node join and leave:** In order to join a Pastry network, a new node must know an existing node. The new node can initialize its state by contacting the existing node by sending a *join* message with its nodeId as the key. The message is routed to another existing node with nodeId numerically closest to new node's nodeId. Then all nodes encountered on the routing path send their state tables to *X*. The new node *X* then initializes its own state tables based on the new information. Finally, the new node informs any nodes that need to be aware of its arrival. Routing table maintenance is handled by periodically exchanging keep-alive messages among neighbor nodes.

Upon detecting node failure, all members of the failed node's leaf set are then notified and they update their leaf sets.

### 3.3. Kademlia

The basic principle of Kademlia [10] is to successively find nodes that are half the distance to the target node. Kademlia differs from Pasty and other such overlays in mainly two different aspects. One difference is a new notion of node closeness based on XOR of the node identities. The other difference is Kademlia nodes contain lists of entries, referred to as buckets, which are used to send parallel requests.

**Address Space:** Kademlia system assigns 160-bit node IDs. The lookup algorithm uses a XOR-based closeness to reduce the lookup space. The intuition behind the XOR based closeness is that node IDs that are different at higher order bits matter more than node IDs that are different in lower order bits and hence, the XOR distance would be higher. Using this XOR metric, Kademlia's topology orders nodes as a tree where subtree nodes are closer together than other subtrees.

**Routing table and key lookup:** Routing tables contain separate lists for each bit in the node ID. Hence if a network uses 128 bits for node IDs each node will have 128 lists (called buckets in Kademlia). Each list corresponds to a particular distance to nodes. Distance is measured in matching bits in the node IDs. Nodes in the nth list have a differing nth bit from the current node's ID whereas the first n-1 bits match those of the current node's ID. To define distance between nodes, Kademlia uses XOR metrics. Here the result of the XOR operation applied to two node IDs (returning 0 for identical bits and 1 for differing bits) is the distance between two nodes. Like Chord, Kademlia nodes know about more nodes near to them and fewer nodes further away.

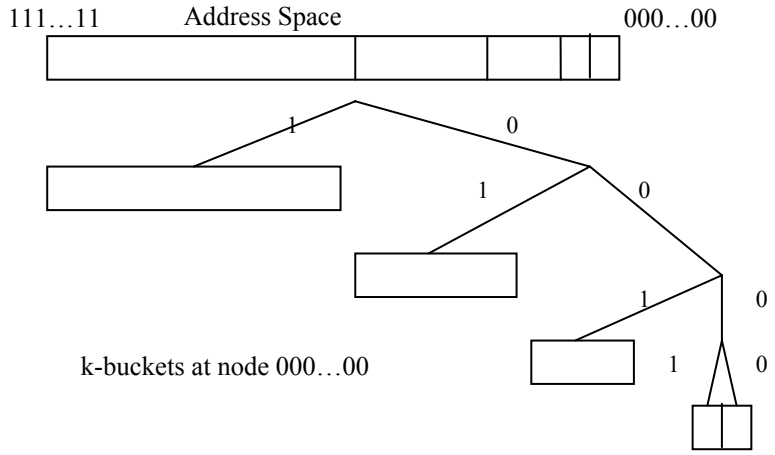


**Figure 4: Kademlia Routing Table Data Structure.**

Figure 4 shows a routing table for a node with ID 000..00. Note that there are k-buckets, each of which covers an address space based on the XOR metric of node IDs. Each of these buckets is a list that may contain multiple contacts for a given subtree. Maintenance of these buckets is straightforward though highly unbalanced trees are handled separately. Maintenance of the nodes in the lists could be dependent on the applications.

A Kademlia lookup node first finds the k-closest nodes to the given node ID. Kademlia recursively picks a subset, l, of these k-closest nodes and sends a request to all the l nodes. In the next recursive step, the Kademlia lookup node again picks a subset of nodes from the nodes it learned about from the previous request. Intuitively, each of these recursive reduces the XOR metric distance by ½ and results in the smaller size k-buckets. The concurrent lookup provides a trade off between bandwidth and lookup latency.

**Node join and leave:** Node joins mirror node lookups. That is, a node, u, that wishes to join adds a previously known contact, w, to its bucket and performs a node lookup. It fills up its routing table based on the responses and inserts itself into the k-buckets of the

other nodes in the system. There is no specific mechanism for node departures as other nodes may discover through the PING mechanism.

### 3.4. Tapestry

Tapestry [31] was developed by a team of researchers from University of California, Berkeley, and MIT. Tapestry has close links with Pastry in that both offer a prefix-based routing of messages. Tapestry aims at providing high-performance, scalable and location-independent routing of messages to near-by endpoints. Tapestry exploits locality when routing messages, including object replicas. Especially, Tapestry allows applications to place object replicas according to the application's need. Bayeux [32] as a Application Lay Multicast approach has been implemented on Tapestry. Chimera [33] is a more recent and updated Java-based implementation which uses Tapestry concepts.

**Address Space:** Tapestry nodes are assigned node IDs uniformly at random from a large identifier space. Typically a 160 bit values are used together with a globally defined radix. Usually the radix is defined as hexadecimal resulting in 40-digit identifiers. The SHA-1 hashing algorithm may be used to create node IDs. Data items (or Application specific endpoints) are assigned unique identifiers from the same ID space.

**Routing table and key lookup**: Each node maintains a routing table whose entries consists of node IDs and the corresponding IP addresses. All nodes represented in a routing table are called 'neighbours' of that node. Routing corresponds to forwarding messages across neighbour links to nodes which are closer, i.e. matching more digits of the prefix, to the key of the endpoint. An example routing table is shown in Figure 1. This routing table belongs to node 3176 in an overlay which uses 4-digit octal IDs. Each routing tables has a number of levels corresponding to the number of digits used in the IDs. For the example shown in Figure 5 this corresponds to 4 levels. Each level contains links to nodes matching a prefix up to a digit position in the ID. Each level contains a number of entries

equal to the radix used (in Figure 5, 8 as octal). Further, the primary *i*th entry in the *j*th level corresponds to the closest node whose ID begins with the corresponding prefix. Using this 'closest node' approach provides the locality properties of Tapestry.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0XXX | | 30XX | | 310X | | 3170 | |
| 1XXX | | -- | | 311X | | 3171 | |
| 2XXX | | 32XX | | 312X | | 3172 | |
| -- | | 33XX | | 313X | | 3173 | |
| 4XXX | | 34XX | | 314X | | 3174 | |
| 5XXX | | 35XX | | 315X | | 3175 | |
| 6XXX | | 36XX | | 316X | | -- | |
| 7XXX | | 37XX | | – | | 3177 | |

**Figure 5: Routing Table example for Node ID 3176.**

Each hop in the routing of a message takes the message closer to its destination. Specifically, the node for the *n*th hop shares a prefix of at least *n* digits with the destination ID. This approach guarantees that any node in the system can be reached in at most $\log_\beta N$ overlay hops, where $N$ is the size of the namespace and $\beta$ is the radix used.

**Node Join and Leave:** Inserting a new node N starts at the node that is responsible for the ID of N in the overlay. This surrogate node S determines *p*, the number of digits its ID shares with N's ID. S then sends a multicast message to all nodes which share the same prefix. These nodes will add N to their routing table and in turn contact N, so N can add these nodes to its own routing table. N then carries out an iterative nearest neighbour search starting at level p. N may trim the list to the closest *k* nodes. N then requests these p nodes to send their backpointers at that level. This results in a set of all nodes that point to any of the *k* nodes at the previous routing level. Next N decrements *p* and repeats the process for all remaining levels.

If a node N decides to leave the network, it notifies all nodes in N's backpointers about it leaving. With each notification N provides a replacement node from its own routing table. Any object references stored on N are rooted to their new hosts. Nodes that left the network ungracefully are detected using periodic beacons. Such leaving

events trigger repair of the overlay and initiate redistribution of object references.

### 3.5. P-Grid

P-Grid [34][35][36] uses a virtual binary tree to form an overlay. The virtual tree is used to distribute the data items to be stored in the overlay to one or more peers. P-grid achieves O(log n) performance for search operations where n is the number of data items in the overlay.

**Address Space:** P-Grid is similar to Kademlia in that it also uses tree-based routing. However, in P-Grid the node IDs are disentangled from the key IDs. In fact, there is no requirement to hash node IDs in P-Grid. Due to the nature of the construction of key IDs, P-Grid supports substring queries. This is probably one of the most distinguishing features of P-Grid when compared with other DHT based overlays. Like some other DHTs Pastry, Tapestry), P-Grid uses a prefixed based search algorithm.

**Routing table and key lookup:** Each peer in P-Grid contains a routing table which contains an entry for every bit in the binary tree. The example shown in Figure 6 uses a 3-bit binary tree. Hence each routing table has 3 entries. Each entry corresponds to a bit in the path towards that node and stores at least one peer that is responsible for the other side of the binary tree at that level. As an example Node 6 is linked to 011 in the binary tree. Hence it has entries for the other side of the tree at the topmost level, in this case 1 (Node 5), the second level, in this case 00 (Node 4), and the bottom level of the tree, here 010 (Node 2).

The binary search tree can be constructed for any set of strings. To construct a tree a sample search string database is used. Firstly, the length of the common prefix of the strings in the database is calculated. This database is lexicographically sorted, and the string at the middle position in the sorted database is selected. The prefix of this string (length is the common prefix + 1) is determined and used to split the database in two equally sized parts. The prefix is then stored

at the root of the tree. This splitting proceeds until the desired depth of the tree is achieved. The binary key of a string is then calculated by comparing the string to the tree's root value. If the string is smaller than this then 0 is appended to the key and the left subtree is considered next, and if it is greater then 1 is appended to the key and the right subtree is considered next. This algorithm is carried out for every level in the tree.

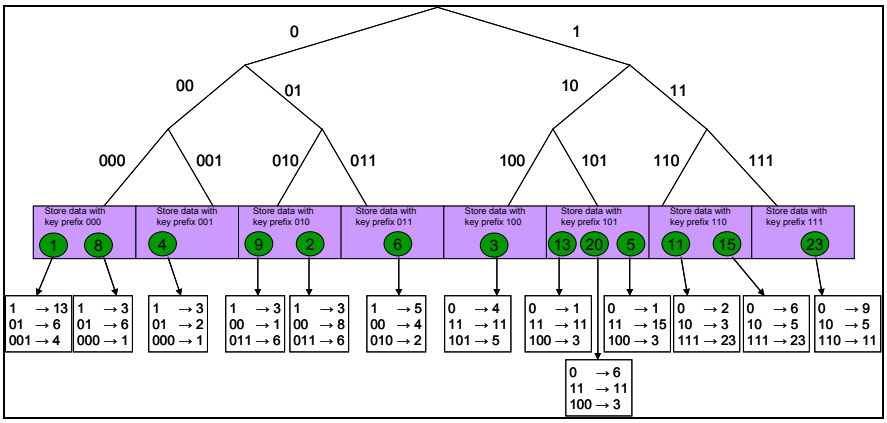


**Figure 6: Example P-Grid with a 3-bit binary tree.**

In P-Grid a number of nodes might be responsible for the same part of the tree. For instance in Figure 6, Nodes 1 and 6 are responsible for data with the prefix 000, Nodes 9 and 2 are responsible for data with the prefix 010, Nodes 13, 20, and 5 are responsible for data with the prefix 101 and Nodes 11 and 15 are responsible for data with the prefix 110. If a peer receives a query it cannot directly satisfy it will forward the query to a node closer to the destination. For instance, in Figure 6, if Node 4 receives a query for 101, it will forward the query to Node 3 as the query starts with 1. Node 3 will forward the query to Node 5 which is responsible for this data and will return it to the original requesting node.

**Node Join and Leave:** P-Grid construction is carried out by local interactions between peers only. It is assumed that by some mechanism peers will meet and interact. Initially, all peers are responsible for the entire search space, i.e. all keys. As two peers meet, they di-

vide the search space into two halves, with each node taking responsibility for one half. This approach is carried out whenever two peers meet which have responsibility for the same address space. If peers meet, which are responsible for data items whose keys have a common prefix, they can initiate additional meetings by forwarding each other to peers in their respective routing tables. If the meeting peers have a different path length the peer with the shorter path can specialise by extending its path in the opposite direction from the other peer at that level. This algorithm is uniform and self-stabilising.

# 4. Constant Degree Overlays

## 4.1. CAN

CAN, or Content Addressable Network [11], is one of the first DHTs (distributed hashing table) proposed.

**Address Space:** CAN utilizes a virtual *d*-dimensional Cartesian coordinate space to host both keys and nodes. Keys and nodes are mapped to corresponding points/coordinates in this *d*-dimensional space using a uniform hash function. Hence the address of a node is its location within the logical coordinate system. As with all DHT based algorithms, the location of a node is calculated using a hash function. Then, the entire coordinate space is divided into "zones" where each node owns one zone. Node that owns the zone is responsible for the keys sitting in the same zone.

**Routing table and key lookup:** A CAN node's routing table contains coordinates and IP addresses of each of its neighbours in the coordinate space. Neighboring nodes are the nodes whose zones adjoin each other. In a *d*-dimensional space, two zones adjoin if their coordinate spans overlap along *d-1* dimensions.

The key lookup starts with hashing the search key to a coordinate in *d*-dimensional space. A CAN message is then formed carrying the destination coordinates. The lookup message is forwarded toward the destination zone in a greedy fashion: the node always forward

the message to its neighbor node that is closer to the destination. A tie is broken arbitrarily. Figure 7 illustrates such an example in a 2-dimension CAN space.

For a d-dimensional CAN space with n nodes, the average routing path length is $(d/4)(n^{1/d})$ hops. The routing table size is $2d$. Hence the routing table size is independent of number of available CAN nodes, and the routing path length grows in the order of $O(n^{1/d})$.

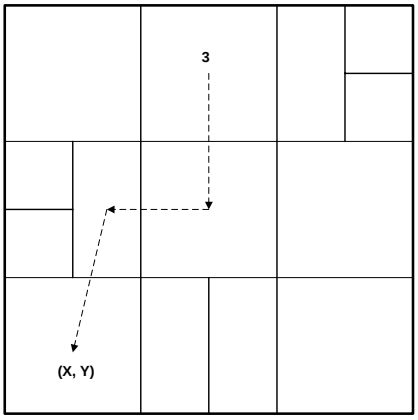


**Figure 7: A 2-dimension CAN space with 14 zones. Node 3 initiates a lookup for key (X,Y) using greedy algorithm**

The routing table in CAN is maintained through periodic update messages. As long as the routing table contains the right neighbors, the lookup service will succeed. In case a node loses multiple entries in the routing table simultaneously, or the rebuilding process has not fully recovered the routing table, a node may use stateless, controlled flooding to locate a node closer to the destination. The closer node then takes over the lookup and uses greedy forwarding thereafter.

**Node join and leave:** A newly arrived node knows at least one existing node in CAN. It randomly generates its coordinate, *P*, in the virtual space. A JOIN request with destination *P* is sent through the known CAN node. Once the destination CAN node receives the JOIN request, it will split its zone into half and assign one half to the new node.

The new node builds up its routing table through learning previous owner node's routing table. The routing table comprises of the subset of retrieved routing table plus the occupant node as neighbor. The routing tables of neighboring nodes also need to be updated. The new and previous occupant nodes send out update messages to neighboring nodes. In fact, the zone information of a node is periodically sent to a node's neighbors, which ensures the correctness of routing table.

When a node leaves the system gracefully, its zone and associated *(key, value)* database is handed over to one of it neighbors. The new zone owner merges the handed over zone with its original zone.

In case a node fails or departs unexpectedly, the periodical update message will discover such departure/failure. The takeover mechanism is automatically trigged at nodes that discover the failure. The takeover timer is started. When the takeover timer expires, the node sends a TAKEOVER message with its own zone information to all of the failed node's neighbors. The receiving node either cancels its own timer if the zone volume in the message is smaller than its own zone volume, or it replies with its own TAKEOVER message. The goal is to have the neighbor node that has the smallest zone to take over the orphan zone.

### 4.2. Ulysses

Ulysses [37] can achieve tries to achieve $log_2 log_2 n$ end-to-end routing latency with a routing table size of about $log(n)$.

**Address space:** The structure of Ulysses is based on the butterfly topology [38], but it improved the static butterfly topology by accommodating the dynamics of peer-to-peer networks. In addition, it solves the problem of high edge stress of static butterfly topology by adding *shortcut links.*

Figure 8 shows a Ulysses network with 2 levels and 11 nodes. In a Ulysses network, a node is identified by a tuple *(P, l)*, where *l* is the

level number and *P* is a binary string uniquely identifying the node in the level. *P* can be mapped to a *k*-dimensional row identifier ($x_0$, $x_1$, …, $x_{k-1}$ ) in a static butterfly is as follows : The bits at location *i, i + k, i + 2k, ...* in *P* represent $x_i$ in ($x_0$, $x_1$, …, $x_{k-1}$ ). The length of *P* in a Ulysses network with *n* nodes and *k* levels is expected to be $log_2(n/k)$. But the length of *P* for individual nodes changes due to dynamic arrival and departure of nodes.

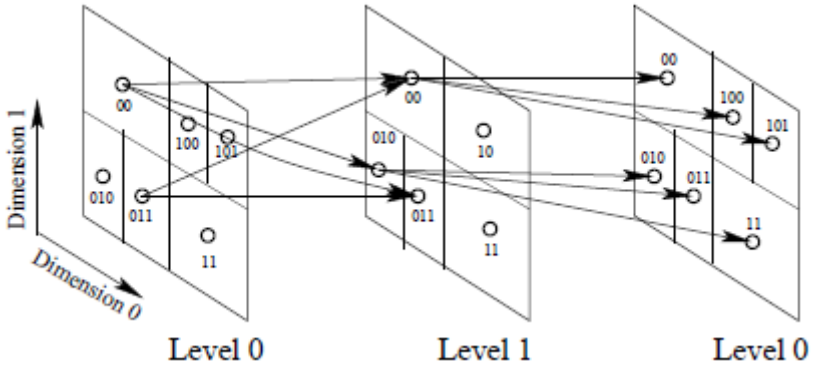


**Figure 9: A Ulysses butterfly with 2 levels and 11 nodes.**

**Routing table and key lookup:** In a Ulysses network with *k* levels, a query for the key *(α, i)* originates at a random node. The query keeps getting forwarded to next level. In each forwarding step, the forwarded node can match one additional dimension of the key. After the *k* steps the query reaches a node *(Q, l)* such that *α* lies within the zone Q in all the *k* dimensions. If the level *l* is the same as the level *i* of the key that is being searched, then *Q* must contain α, and the routing is complete. Otherwise the node *(Q, l)* forwards the query on its shortcut link to node *(Q, i)*, which must be responsible for the key *(α, i)*.

**Node join and departure:** A new node must know an existing node to join a Ulysses network. It then generates a random key and sends a query for this key through the existing node. This query will eventually reach the node O *(Q,l)* responsible for the key. Node *O* then splits its zone of responsibility in two and assigns one half to the new node. The identifiers of node *O* and the new node will then be *Q0* and *Q1*, respectively. Both nodes remain in level *l*. The node *O*

informs the new node *N* about its original neighbors. When a node with identifier *(P, l)* leaves the network, it needs to hand over its keys to another node at the same level.

### 4.3. Cycloid

Cycloid [39] presents an overlay that combines hypercube routing with overlay routing that reduces the lookup path by key matching as in Pastry. Cycloid uses Cube-Connected-Cycles graph as its geometry. The geometry and the routing algorithm ensure that the lookup time is O(d), where 'd' is the network dimension.

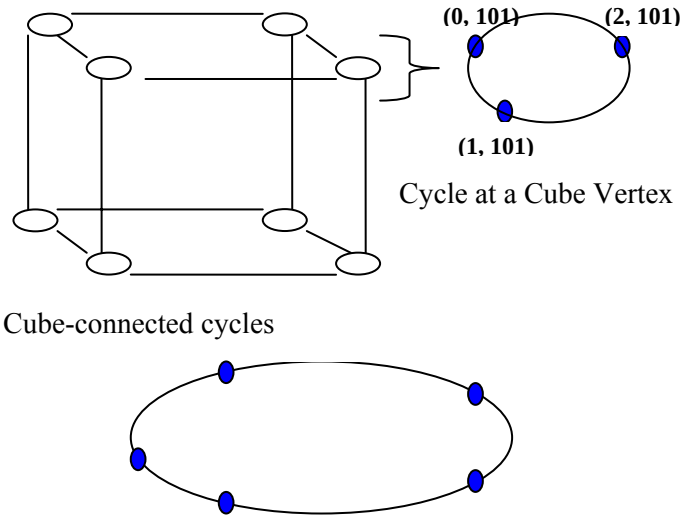


**Figure 10: Various linkages of a full 3-Dimensional Cycloid.**

**Address Space:** A cycloid can be viewed as a d-dimensional cube where each vertex is replaced by a cycle of d-nodes, with $n = d.2^d$, where n is the number of nodes in the network. The connectivity of each node is constant and the finger table size (discussed later) is constant. Each node in a cycloid is represented as a tuple (k, $x_{d-1}$, $x_{d-2}$, …, $x_{0)}$ with k as the cyclic index and $x_{d-1}$, $x_{d-2}$, …, $x_0$ as the cubical index. The cyclical index is an index between 0 to d-1 and the cubical index is binary number between 0 and $2^d$ - 1.

Figure 10 shows various linkages of a 3-dimensional cycloid. The cube shows how cycles at different levels are connected. Each cycle connects nodes that have the same cubical index but with different cyclical index. Finally, nodes with different indices are connected as a large ring that enables nodes on cycle to reach nodes in another cycle directly or indirectly.

To minimize the finger table size and maintenance, each node in a ring is connected only to a primary node with highest cyclical index in its preceding cycle and succeeding cycle. The predecessor and the successor of a node, say (k, $x_{d-1}$, $x_{d-2}$, …, $x_0$), in such a ring are chosen such that the most significant different bit (MSDB) with the current node is no larger than k - 1. The predecessor and successor are chosen such that they are the first such largest and first such smallest nodes. Note that this arrangement of nodes with different cubic indices as a ring gives cycloid a lookup ability to select a cubical index that is closest to its destination from different cycles. Within each cycle, a node is connected to its predecessor and successor nodes. Hence the finger table of a node has seven entries and has the following entries.

| | **Node ID (k,** , $x_{d-1}$, $x_{d-2}$, …, $x_0$) |
|---|---|
| 1 | **Cubical Neighbour --** **(k – 1,** $x_{d-1}$, $x_{d-2}$, …, $x_k$, x x x x) |
| 2 | **Cyclic Neighbour --** **node at k – 1 with max cyclical index less than** $x_{d-1}$, $x_{d-2}$, …, $x_k$, x x x x |
| 3 | **Cyclic Neighbour --** **node at k – 1 with min cyclical index greater than** $x_{d-1}$, $x_{d-2}$, …, $x_k$, x x x x |
| 4 | **Inside Leaf set predecessor** |
| 5 | **Inside Leaf set successor** |
| 6 | **Outside Leaf set predecessor – primary node of preceding cycle** |
| 7 | **Outside Leaf set successor -- primary node of succeeding cycle** |

**Table 1**: Routing Table of a Cycloid node

Cycloid key assignment consists of generating a pair of cyclic and cubic indices. For a give key, the cyclic index is its hashed value modulated by d and the cubic index is the hash value divided by d.

**Routing table and key lookup:** The routing algorithm of a cycloid DHT consists of three steps. The first step uses the outside ring or the outside leaf sets of the finger table to find out the closest cubical

neighbour or the closest cyclical neighbour. Then the inside leaf sets are used to find appropriate node. The following steps are performed by a source node (k, $x_{d-1}$, $x_{d-2}$, …, $x_0$) to route to a destination node (l, $y_{d-1}$, $y_{d-2}$, …, $y_0$). MSDB represents the most significant different bit between the source and the destination nodes.

1. Ascending: If k < MSDB then it forwards request to a node in the outside leaf set until k >= MSDB. This step helps in either finding the closest cubical neighbour or closest cyclical neighbour.
2. Descending: If k=MSDB, then request is forwarded to the inside cubical neighbour else if k > MSDB then the request is forwarded to the closest cyclical neighbour.
3. Traverse Cycle: If the target ID is within the inner leaf set, then inside leaf set entries from the finger table are used for lookup.

**Node Join and Leave:** A joining node X will route the joining message through a bootstrapping node to a node Y whose ID is numerically closest to X. The finger table of X forms its leaf sets based on the finger table of Y.

1. If X and Y are in the same inside leaf set then the outer leaf set values of X are the same as Y. The inside leaf set values of X and Y are modified according to the position of X with respect to Y and others in the inner leaf set.
2. If X is the first in its cycle, then it has to form the links to the outside leaf sets. If Y's cycle is the succeeding remote cycle of X, then Y's left outside leaf node and primary node are the left and right nodes in X's outside leaf set. Otherwise the right outside leaf and the primary node are used. Since X is the only node in its cycle, its inside leaf sets point to itself.

Once a node joins, it propagates the join information to its entries which update themselves. Inner leaf sets update themselves while the outer leaf sets update themselves and propagate the join to their inner leaf sets.

When a node is leaving, it notifies inside leaf set nodes. If it is a primary node then the leaving node has to update its outside leaf

sets. Upon receiving such message, the outside leaf set nodes update themselves and transfer the leave notification to its inside leaf set. The cycloid DHT leaves the updating of cubical and cyclical neighbours of leaving nodes and of failed nodes as the responsibility of system stabilization.

# 5. O(1)-Hop Overlays

## 5.1. Kelips

Kelips [40] is based on a DHT overlay that uses increased memory and increased background overhead for efficient O(1) lookup. Kelips overlay is simple and differs from other structured P2P overlays in mainly two ways. One difference is that it is loosely structured and the other is that the loose structure does not preserve any invariant. However, Kelips uses increased memory and sophisticated broadcast mechanism to achieve a reliable O(1) lookup. Hence, though Kelips lookup, join, and leave mechanisms are quite simple they rely on sophisticated broadcast mechanisms among its members.

**Address Space:** Kelips consists of k virtual affinity groups that are formed by hashing a node's identifier. Each node's finger table consists of a set of other nodes in its affinity group, a set of nodes in all the foreign groups, and a set of file tuples that give details of a file and the id of the node storing the file. Hence the total storage requirements of Kelips could be of the order $n/k + c \times (k - 1) + F/n$, where n is the number of nodes, k is the number of affinity groups, c is the contact size, and F is the file size.

There is no geometry to the address space and each node knows about a larger set of nodes. Kelips relies on a gossip-style epidemic [41][42] protocol to ensure that with high probability the finger table information is transmitted to all nodes. To ensure this, nodes in Kelips overlay use a light weight protocol to transmit limited information, such as keep alive packets and filetuple information, to nodes in their affinity group and contacts group. These nodes in turn chose other nodes from their finger tables to propagate such information.

There are studies that show that such a gossip protocol is quite robust against packet losses and node failures [43][44].

**Routing table and key lookup:** A querying node maps a file name to appropriate affinity group and sends a look up request to the topologically closest node in that affinity group. The receiving node looks up its table and returns to the querying node the filetuple with the address of the *homenode* storing the file. The querying node then requests the homenode directly for the file.

Nodes that wish to insert a file follow the same procedure. They send the request to the topologically closest node from the hashed affinity group. The receiving node randomly picks one node from its affinity group and assigns the file to it and designates it as the homenode. The homenode then inserts the file, creates a new filetuple and inserts that information into the gossip stream.

**Node Join and Leave:** A bootstrapping node allows the joining node to create a soft finger table and allows it to join the gossip stream. Since there is no structure or invariant that Kelips has to preserve, join is complete with the node participating in the gossip stream. Node leaving or failures are updated through out the system through the gossip mechanism. If other nodes notice the lack of updates from the failed node, they update their entries accordingly.

### 5.2. OneHop

OneHop [45] was developed by a team at MIT. Some members were also involved in the development of Chord and EpiChord. So in many aspects OneHop relates to these two overlay algorithms.

**Address Space:** OneHop nodes are assigned a 128-bit random node ID. These IDs are ordered in a ring modulo $2^{128}$. Identifiers are uniformly distributed as can be achieved by using hash functions such as SHA-1. Like in Chord, every node has a predecessor and a successor in the identifier ring. Each node periodically sends keep-alive messages to its predecessor and successor. Data items are assigned

an ID in the same ID space. The node which should store a data is the successor, i.e. the first node in the ring clock-wise from the key.

**Routing table and key lookup**: In OneHop, every node maintains a full routing table, that is a routing tabe contains references to all other nodes in the network. This is to allow sending a lookup request from the source node to the destination node in a single hop. The source node will look up the successor node of the data key is requires, and send a lookup message to this node. Key to this scheme is that the routing tables are up-to-date. Hence node leave and join events need to be propagated to all nodes.

**Node Join and Leave:** Join and leave events need to be send to local nodes, but also to all the other nodes in the overlay. Local updates include updates to successor and predecessor nodes. Every node n runs a stabilisation routine periodically, which involves sending keep-alive messages to its successor s and predecessor p nodes. Node s checks if n is indeed its predecessor. If not, it informs n that there is another node between them. Similarly, p checks if n is its successor, and if not it notifies n. If either s or p do not respond, n will ping them repeatedly and after a timeout interval conclude that the node is dead.

A joining node contacts any other node in the overlay and gets its routing table, similarly to the approach in Chord. With this information the new node can establish its successor and predecessor and inform them of its existence.

Both join and leave events also need to be forwarded to all the other node in the system within a certain time. This is achieved by imposing a hierarchy on the system, forming a tree. This hierarchy is introduced by dividing the 128-bit identifier space into k equal contiguous intervals (slices). All slices will have roughly the same number of nodes as nodes have uniformly distributed random identifiers. Each slice has a slice leader which is the successor of the midpoint of the slice identifier space. New nodes learn about their slice leader from one of its neighbours. Slices are again divided into equal-sized intervals (units). Each unit has also a unit leader which

is the successor of the mid-point of the unit identifier space. As a node detects a change in the membership of the overlay it informs its slice leader. The slice leader aggregates notifications from its members for a certain interval before sending them out to other slice leaders. The slice leaders again aggregate notification messages for a certain interval before sending them on to the unit leaders within their slice. Unit leaders piggy back these update information on keep-alive messages to their successor and predecessor nodes. Other nodes propagate this information also via keep-alive messages – if they receive the information from their predecessor they forward it on to their successor and vice versa, but not beyond unit boundaries preventing duplicate messages. If a slice leader fails, this will be detected by its successor, and this node will become the new slice leader.

### 5.3. EpiChord

EpiChord [9] is a variation of Chord that intends to speed up the lookup process. The speedup is achieved using parallel queries and by allowing nodes to cache more routing entries than *O(log N)* entries in original Chord protocol. EpiChord is able to achieve *O(1)-hop* lookup performance under lookup-intensive workloads, and at least *O(log N)*-hop lookup performance under churn-intensive workloads in the worst case.

Same as in Chord, the nodes and keys are mapped to *m*-bit identifiers using consistent hashing. Nodes are ordered on an identifier circle modulo $2^m$. A key is cached at its successor node. The successor node to a node or key is the next node in the identifier circle in the clockwise direction; while the predecessor node to a node or key is the next node in the identifier circle in the counter-clockwise direction.

**Routing table and key lookup:** To guarantee the routing correctness in the face of peer churn, each EpiChord node maintains a list of *k* successor nodes and *k* predecessor nodes, respectively. Furthermore, EpiChord divides the address space into two half circles, with each half circle being further divided into a set of exponentially

smaller slices (see Figure 11). It is required that at least *j/1-q* entries are maintained in the routing table for individual slices, where *j* is pre-determined number of entries per slice, and *q* is the probability that a entry is out-of-date. Since EpiChord node maintains *k* successor nodes and *k* predecessor nodes and both should fit in two smallest slices, the number of slices can be estimated. Further, the parameters *j* and *k* satisfy the following relationship, $k \geq 2j$.

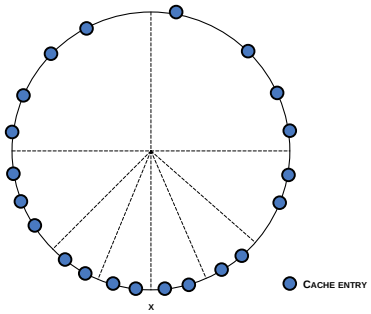


**Figure 11: Division of address space into exponentially smaller slices with respect to node *x*.**

The key lookup process utilizes *p* parallel queries. For a given key *k*, the node selects one node immediately succeeding *k* and *p-1* nodes preceding *k* from its routing table. Upon receiving the search query, if the probed node owns the key, it responds as the destination node and the searching process finishes. If the probed node is not the destination node, it returns *l* best next hops from its routing table. In addition, the probed node returns its immediate successor if it is a predecessor of key *k*, or its immediate predecessor if it is a successor of key *k*. Both *p* and *l* are configuration parameters. When these replies are received by the original node, further queries are dispatched if returned nodes are closer to the key *k* than the nodes that have already responded. Notice that only original node issues queries and the lookup proceeds in an iterative fashion.

In EpiChord, each entry in the routing table is associated with a lifetime. Routing entries are flushed whenever lifetime expires. In addition, the routing entry associated with a node is purged if the node does not respond to some number of queries.

EpiChord nodes also monitor the number of entries available at each slice. Should a slice be found not to have sufficient routing entries, a node makes a lookup to the midpoint of that slice, and increases routing entries from the lookup responses.

**Node join and leave:** A new node knows at least one node already in EpiChord. It sends queries to this node. Since EpiChord nodes constantly update their routing tables by observing lookup traffic, other nodes learn about this node eventually. Besides, the new node obtains a full cache transfer from one of its two immediate neighbors.

EpiChord employs stabilization strategy to resolve the routing table inconsistency problem when multiple nodes join EpiChord at about the same location, or nodes leave the system unexpectedly. The stabilization strategy comprise of weak stabilization protocol and strong stabilization protocol.

- *Weak stabilization protocol:* Weak stabilization protocol maintains weak stable relationship among nodes, i.e., *predecessor(successor(n))=n*. To achieve this, nodes periodically probe their immediate neighbors to check if they are alive. It is individual node's responsibility to maintain its successor and predecessor. When a node with closer Id than a node's successor or predecessor is discovered, either through observing lookup traffic, or through active probing neighbors, the node update its successor or predecessor, correspondingly.
- *Strong stabilization protocol:* for a weak stable EpiChord ring, it is still possible the ring is loopy. Strong stabilization protocol ensures that the loop will be detected and fixed. The basic idea of loop detection is to let a query traverse the entire ring. If a loop exists, the traversal allows EpiChord nodes to know nodes whose Ids are closer than its successor or predecessor, thus break the loop by updating successor/predecessor nodes.

### 5.4. D1HT

D1HT [46] was first introduced in 2005 by researchers from Federal University of Rio de Janeiro, Brazil. The design goal of the overlay is to maximize performance with reasonable maintenance traffic overhead even for huge and dynamic peer-to-peer (P2P) systems. The philosophy of D1HT design is that the tradeoff between latency and bandwidth usage should favor latency because speed and information are critical while network bandwidth improves over time.

**Address space:** A D1HT system is composed of a set ***D*** of $n$ peers and maps items (or keys) to peers based on consistent hashing, where both peers and keys are hashed to integer identifiers (IDs) in the same ID space $[0..N]$, $N >> n$. Typically a key ID is the cryptographic hash SHA-1 of the key value, a peer ID is based on the SHA-1 hash of it's IP address (or the SHA-1 hash of the user name), and $N = 2^{160} - 1$. As in Chord, D1HT uses a ring topology where ID $0$ succeeds ID $N$, and the successor and predecessor of an ID $i$ are respectively the first living peers clockwise and counterclockwise from $i$ in the ring.

**Routing table and key lookup:** Each peer in a D1HT system maintains a routing table with the IP addresses of all peers in the system, and so any lookup is trivially solved with just one hop, provided that the local routing table is up to date. As each peer in a D1HT system should know the IP address of every other peer, any join/leave events should be acknowledged by all peers in the system in a timely fashion in order to avoid stale entries in routing tables.

**Node join and departure:** A new node must know an existing D1HT node to join a D1HT system. The joining peer hashes its IP address (or some other unique value) to get its ID $p$ and asks the existing node to issue a lookup for its ID. The query will return $p$'s successor. The joining peer then contacts its successor to join the network and get the information about the keys it will be responsible for. The successor will also send the IP addresses of a number of peers to the new node. The new node will then ping those peers and choose the nearest ones to get the routing table. D1HT also uses a *Quarantine* mechanism to handle highly dynamic nodes, where a joining peer will not be granted to immediately take part of the

D1HT overlay network, though it will be allowed to perform lookups at any moment. When a node leaves, instead of sending the leaving event to all the other peers, D1HT uses the EDRA (Event Detection and Reporting Algorithm) algorithm to propagate the event. The details of EDRA algorithm can be found at [46]. A study [47] found some shortcomings with the EDRA algorithm and termed the updated version EDRA*.

## 6. Comparison and Analysis

In this section we compare and contrast different structured P2P overlay technologies from the geometries used, the routing algorithms, routing performance, join/leave efficiency, routing table maintenance, and bootstrapping strategy. The results are summarized in the Tables below.

| | **Chord** | **Pastry** | **Kademlia** |
|---|---|---|---|
| **Geometry** | Circular Node-ID space, logarithmic degree mesh | Plaxton-style mesh network, prefix routing | XOR metric for distance between points in the key space. |
| **Routing algorithm** | Search query forwarded to "closer" node | Matching Key and prefix of Node-ID | (XOR) Matching Key and Node-ID based routing done parrallely. |
| **Routing performance** | $O(logN)$, where $N$ is the number of peers | $O(log_BN)$, where $N$ is number of peers, and $B=2b$, $b$ is number of bits of NodeID | $O(log_BN)+c$, Where $N$ is number of peers, $B=2b$, $b$ is number of bits of Node-ID, and $c$ is a small constant |
| **Join/Leave performance** | $(logN)^2$ | $log_BN$ | $log_BN+c$ |
| **Routing table maintenance** | Periodic stabilization protocol at nodes to learn about newly joined nodes, update successor and predecessor, and fix finger tables. | Neighboring nodes periodically exchange keep-alive messages. The leaf sets of nodes with adjacent Node-Id overlap. | Failure of peers will not cause network-wide failure. Replicate data across multiple peers. |
| **Bootstrapping** | A new node knows an existing Chord | A new node knows a nearby Pastry | A new node knows an existing Kadem- |

| | | | |
|---|---|---|---|
| | node. | node. | lia node. |

**Table 2: Summary of Chord, Pastry and Kademlia overlays.**

| | **Tapestry** | **P-Grid** | **CAN** |
|---|---|---|---|
| **Geometry** | Uniformly at random from a large identifier space (typically 160bit with a globally defined radix) | Binary tree | *d*-dimensional Cartesian coordinate space |
| **Routing algorithm** | Matching Key and prefix in Node-ID | Binary tree search and prefix matching | Forward the search message to neighbour node closer to the destination |
| **Routing performance** | $\log_\beta N$, where $N$ is the size of the identifier space, and $\beta$ is the radix used | *O(log N),* where $N$ is number of data items in the overlay) | $(d/4)(N^{1/d})$, $N$ is number of nodes, and $d$ is dimension |
| **Join/Leave performance** | | *O(log N)* | *2d* |
| **Routing table maintenance** | | | Through periodic update messages. Controlled flooding is used in case a node loses multiple entries simultaneously |
| **Bootstrapping** | A new node knows a nearby Tapestry node. | Know at least one node. | Know at least one node. May get this node through DNS |

**Table 3: Summary of Tapestry, P-Grid and CAN overlays.**

| | **Ulysses** | **Cycloid** | **Kelips** |
|---|---|---|---|
| **Geometry** | Butterfly topology with shortcut links | Cube-Connected-Cycles graph | No geometry to the address space, each node knows about other nodes |
| **Routing algorithm** | For a Ulysses network with $k$ levels and $n$ nodes. to search a key $\alpha$, in each step, the query gets locked in one additional dimension, after the first $k$ steps the query reaches a node $(Q, l)$ such that $\alpha$ lies within the zone Q in all the $k$ dimensions. | Uses the outside leaf sets of the finger table to find closest cubical neighbour or the closest cyclical neighbour. Inside leaf sets are used to find appropriate node. | Routing table includes nodes in the affinity group, nodes in all the foreign groups, and filetuples. Querying node maps file name to affinity group and sends lookup request to topologically closest node in affinity group. |
| **Routing performance** | $log_2 log_2 n$ | $O(d)$, where $d$ is network dimension, $n$ is number of nodes, $n = d.2^d$ | $O(1)$ |
| **Join/Leave performance** | Find corresponding node with a randomly generated key, then split the zone. $log_2 log_2 n$ | $O(d)$ | No structure or invariant. Join is complete with node participating in gossip stream. Node leaving are updated through the gossip system. |
| **Routing table maintenance** | | Leaving nodes notify nodes inside leaf set. If primary node then need also update outside leaf sets. Stabilization process detects failed nodes. | Kelips routing tables are maintained through a low bandwidth gossip style mechanism. |
| **Bootstrapping** | A joining node needs to know an existing node in Ulysses network. | No specific bootstrapping mechanism discussed . | Bootstrapping node allowing joining node to join gossip stream. |

**Table 4: Summary of Ulysses, Cycloid and Kelips overlays.**

| | **OneHop** | **EpiChord** | **D1HT** |
|---|---|---|---|
| **Geometry** | 128-bit random node ID ordered in a ring modulo $2^{128}$. | Circular node ID space, logarithmic degree mesh | Hashing keys and peers into an ID space *[0, N]*, *N >> n* |
| **Routing algorithm** | every node maintains a full routing table | Use multiple parallel queries to locate node that owns the key | every node maintains a full routing table |
| **Routing performance** | *O(1)* | *O(1)* under lookup intensive workloads, and *O(log N)* in the worst case | *O(1)* |
| **Join/Leave performance** | | | $\lceil log_2 n \rceil$, where *n* is number of nodes |
| **Routing table maintenance** | Nodes run stabilisation routine sending keep-alive messages to successor and predecessor. | Routing entries are flushed whenever lifetime expires, or the corresponding node does not respond to queries.<br><br>If slice entries are insufficient, lookup to midpoint of slice, add routing entries from the lookup response. | Propagate join/leave messages with TTL values. Use a *Quarantine* mechanism to handle highly dynamic nodes |
| **Bootstrapping** | A new node knows an existing OneHop node. | Knows at least one existing node | The new node must know an existing D1HT peer already in the system |

**Table 5: Summary of OneHop, EpiChord and D1HT overlays.**

# 7. Conclusions

## 7.1. Open Research Issues

We see the future research of structured P2P overlays focussing on the following three aspects, namely algorithms, frameworks, and applications.

1. Algorithms: As discussed in this chapter, many structured P2P algorithms exist. Each of these structured overlays carries its own advantages and disadvantages. The following are key issues that needs to be properly addressed for successful structured P2P algorithms:
    a. Scalability and performance: In general, the structured P2P overlays can help overcome the scalability and performance problems faced by unstructured ones. However, for some real-time applications there are additional performance requirements that need to be addressed, such as Voice over IP (VoIP) and IPTV. Therefore, how to ensure guaranteed performance required by certain applications while keeping the overall system scalable and balanced is an important issue for structured P2P algorithms.
    b. Security, privacy, trust and reputation: Trust and reputation are important to support secured and trustworthy P2P overlay communications among peers. There are many research topics in this area for P2P overlays such as anonymity, denial-of-service attacks, malicious node behavior, reputation and incentives.
    c. Convergence of Peer-to-Peer systems and other established field of distributed computing such as Grid computing.
2. Frameworks: Algorithms theoretically define P2P overlays. In practice, there are many practical issues to be dealt with:
    a. Protocols and interoperability: Peers need to talk to each other. In some scenarios, peers belonging to dif-

ferent P2P overlays may also need to talk to each other. This requires well-defined protocols/interface, and careful study of interoperability among P2P nodes.

b. Heterogeneity: In reality, many aspects can affect the performance of P2P overlays, such as network availability/bandwidth, latency, peers' computational power and storage space, etc. Therefore, supporting heterogeneity is an important issue from a practical point of view.

c. Handle general Internet services: general Internet services, such as spam handling and directory services, are also important to P2P overlays.

3. Applications: Many overlay algorithms and frameworks are developed are with the intention to build novel and useful applications. P2P applications can be in many fields. We just name a few below.

   a. Content sharing/distribution: this category of P2P applications may be the most popularly one so far. There is still plenty of ongoing research work in this area, such as providing better performance, good scalability, fairness, or strong security.

   b. Enterprise applications: P2P applications in enterprise environment typically need to meet more stringent security and performance requirement. It also faces other issues such deployment easiness and monitoring capability.

   c. Communication: For P2P communication applications, the real time constraint (both signaling and media transmission) requires special consideration in P2P algorithm and framework design. In addition, issues such as lawful interception, enabling communication features, and interop with the existing communication networks also demand special attention.

   d. P2P applications in mobile and ad-hoc wireless networks: application of P2P overlay approaches would allow mobile peers to have optimized flow control, load balancing mechanism, and proximity awareness.

e. The semantic grouping of information in peer-to-peer networks: This direction shares many commonalities with efforts in the semantic Web domain.

### 7.2. Summary

Existing products and research projects demonstrate that structured P2P network is an important technology of practical value. It helps overcome the scalability and performance problems faced by many unstructured P2P technologies. This chapter provides an overview of several representative structured P2P overlays, and analyzes and examines key aspects that affect a P2P overlay's performance.

We believe that structured P2P overlay remains to be a viable solution to many problems in distributed computing. There are still many open research questions in this field, such as new algorithms, practical frameworks, and novel applications.

## References

[1] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Content Distribution Technolgies. ACM Computing Surveys, Vol. 36, No. 4, December 2004.

[2] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. IEEE Communications Surveys and Tutorials, Second Quarter 2005, Volume 7, No. 2.

[3] J. Risson, T. Moors. Survey of research towards robust peer-to-peer networks: search methods. Computer Networks 50, 17 (Dec. 2006), 3485-3521.

[4] S. El-Ansary, S. Haridi. An Overview of Structured P2P Overlay Networks. Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks (ed. J. Wu). Auerbach Publications, 2006, pp. 665-683.

[5] A. Gupta, B. Liskov, R. Rodrigues. Efficient routing for peer-to-peer overlays. Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004), 2004, pp. 113-116.

[6] J. Buford, A. Brown, M. Kolberg. Exploiting Parallelism in the Design of Peer-to-Peer Overlays. Journal of Computer Communications, Special Issue on Foundations of Peer-to-Peer Computing. 2008.

[7] M. Kolberg, F. Kolberg, A. Brown, J. Buford. A Markov Model for the EpiChord Peer-to-Peer Overlay in an XCAST enabled Network. IEEE International Conference on Communications (ICC) 2007.

[8] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.* 11, 1 (Feb.2003), 17-32.

[9] B. Leong, B. Liskov, and E. D. Demaine. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. Computer Communications, Elsevier Science, Vol. 29, pp. 1243-1259.

[10] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In Proc of IPTPS02, Cambridge, USA, March 2002.

[11] Sylvia Ratnasamy , Paul Francis , Mark Handley , Richard Karp , Scott Schenker, A scalable content-addressable network, Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, p.161-172, August 2001, San Diego, California, United States.

[12] Krishnamurthy, S., El-Ansary, S., Aurell, E., Haridi, S.: A statistical theory of Chord under churn. In: The 4th International Workshop on Peer-to-Peer Systems (IPTPS'05).

[13] Antony I. T. Rowstron , Peter Druschel, Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, p.329-350, November 12-16, 2001.

[14] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "SCRIBE: A large-scale and decentralised application-level multicast infrastructure", IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No, 8, October 2002.

[15] M. Castro, M. B. Jones, A-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang and A. Wolman, "An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer overlays", Infocom 2003, San Francisco, CA, April, 2003.

[16] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "Scalable Application-level Anycast for Highly Dynamic Groups", NGC 2003, Munich, Germany, September 2003.

[17] Rowstron, A. I., Kermarrec, A., Castro, M., and Druschel, P. 2001. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Proceedings of the Third international Cost264 Workshop on Networked Group Communication* (November 07 - 09, 2001). J. Crowcroft and M. Hofmann, Eds. Lecture Notes In Computer Science, vol. 2233. Springer-Verlag, London, 30-43.

[18] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, June 2002.

[19] M. Castro, P. Druschel, Y. C. Hu and A. Rowstron, "Proximity neighbor selection in tree-based structured peer-to-peer overlays", Technical report MSR-TR-2003-52, 2003.

[20] R. Mahajan, M. Castro and A. Rowstron, "Controlling the Cost of Reliability in Peer-to-peer Overlays", IPTPS'03, Berkeley, CA, February 2003.

[21] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks", SIGOPS European Workshop, France, September, 2002.

[22] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In Proc. HotOS VIII, Schloss Elmau, Germany, May 2001.

[23] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. AC*M SOSP'01, Banff, Canada, Oct. 2001.

[24] S. Iyer, A. Rowstron and P. Druschel, "SQUIRREL: A decentralized, peer-to-peer web cache", 12th ACM Symposium on Principles of Distributed Computing (PODC 2002), Monterey, California, USA, July 2002.

[25] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron and A. Singh, "SplitStream: High-bandwidth multicast in a cooperative environment", SOSP'03, Lake Bolton, New York, October, 2003.

[26] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron and A. Singh, "SplitStream: High-bandwidth content distribution in a cooperative environment", IPTPS'03, Berkeley, CA, February, 2003.

[27] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. Wallach, X. Bonnaire, P. Sens, and J. Busca. POST: a secure, resilient, cooperative messaging system, Proc. of the 9th conference on Hot Topics in Operating Systems, pp. 11-11, Hawaii, May 2003.

[28] Tsuen-Wan "Johnny" Ngan, Dan S. Wallach, and Pete Druschel, "Enforcing Fair Sharing of Peer-to-Peer Resources", IPTPS'03, Berkeley, CA, February, 2003.

[29] Rice University, FreePastry, http://freepastry.rice.edu/

[30] Microsoft, SimPastry/VisPastry, http://research.microsoft.com/en-us/um/people/antr/pastry/download.htm

[31] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, Tapestry: A resilient globalscale overlay for service deployment, IEEE J. Sel. Areas Commun., vol. 22, no. 1, pp. 41-53, Jan. 2004.

[32] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, J. Kubiatowicz , Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination, The 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), Port Jefferson, New York, 2001.

[33] Chimera, http://current.cs.ucsb.edu/projects/chimera/

[34] K. Aberer, M. Hauswirth, M. Punceva, R. Schmidt, Improving Data Access in P2P Systems, IEEE Internet Computing,6(1), January/February 2002.

[35] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, R. Schmidt, P-Grid: A Self-organizing Structured P2P System, SIGMOD Record, 32(2), September 2003.

[36] K. Aberer, A. Datta, M. Hauswirth, P-Grid:Dynamics of self organization processes in structured P2P systems, Peer-to-Peer Systems and Applications, Lecture Notes in Computer Science, LNCS 3845, Springer Verlag, 2005.

[37] A. Kumar, S. Merugu, J. Xu, E. W. Zegura, X. Yu, Ulysses: A Robust, Low-Diameter, Low-Latency Peer-to-peer Network, In European Transactions on Telecommunications (ETT) Special Issue on P2P Networking and P2P Services, 2004. Vol. 15, pages 571-587.

[38] H.J. Siegel, Interconnection Networks for SIMD machines, Computer 12(6), 1979.

[39] H. Shen, C.-Z. Xu, and G. Chen. Cycloid: A scalable constant-degree lookup-efficient P2P overlay network, Journal of Performance Evaluation's Special Issue on Peer-to-Peer Networks (6/29), 2005.

[40] I. Gupta, K. Birman, P. Linga, A. Demers, R. van Renesse. Kelips: building an efficient and stable P2P DHT through increased memory and background overhead. Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS ’03). 2003.

[41] D. Kempe, J. Kleinberg, A. Demers. Spatial gossip and resource location protocols", Proc. 33rd ACM Symp. Theory of Computing (STOC), pp. 163-172, 2001.

[42] R. van Renesse, Y. Minsky, M. Hayden, A gossip-style failure detection service", Proc. IFIP Middleware, 1998.

[43] K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao,M. Budiu, Y. Minsky, Bimodal Multicast, ACM Trans. Comp. Syst., 17:2, pp. 41-88, May 1999.

[44] A. Demers, D.H. Greene, J. Hauser, W. Irish, J. Larson, Epidemic algorithms for replicated database maintenance", Proc. 6th ACM Symposium Principles of Distributed Computing (PODC), pp. 1-12, 1987.

[45] A. Gupta, B. Liskov, R. Rodrigues. Efficient routing for peer-to-peer overlays. Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004), 2004, pp. 113-116.

[46] L. R. Monnerat and C. L. Amorim. D1HT: A Distributed One Hop Hash Table. Proc. of the 20th IEEE Intl Parallel & Distributed Processing Symp. (IPDPS), April 2006.

[47] J. Buford, A. Brown, M. Kolberg. Analysis of an Active Maintenance Algorithm for an O(1)-Hop Overlay, IEEE Globecom 2007.