An Overview of Structured P2P Overlay Networks

Sameh El-Ansary[1] and Seif Haridi[2]
[1] Swedish Institute of Computer Science (SICS), Sweden
[2] Royal Institute of Technology - IMIT/KTH, Sweden

July 19, 2004

Contents

Chapter 1

An Overview of Structured P2P Overlay Networks

1.1 Introduction

Historical Background. The term "peer-to-peer" (P2P) is used in many contexts to mean different things. It was recently used to refer to the form of cooperation that emerged with the appearance of the music file sharing application Napster [31]. With that application, music files where exchanged between computers (Peers) relying on a central directory for knowing which peer has which file. Napster ceased operation due to legal rather than technical reasons and was followed by a number of systems like Gnutella [17] and Freenet, [13] where the central directory was replaced with a flooding process where each computer connects to random members in a peer-to-peer network and queries his neighbors who act similarly until a query is resolved. The random graph of such peers proved to be a feasible example of an overlay network, that is an application-level network on top of the Internet transport with its own topology and routing.

The Motivating Problem. The simultaneous "beauty" and "ugliness" of random overlay networks attracted academic researchers from the networking and the distributed systems communities. The "beauty" lies in the simplicity of the solution and its ability to completely diffuse central authority and legal liability. From a computer science point of view, this elimination of central control is very attractive for - among other things - eliminating single points of failure and building large-scale distributed systems. The "ugliness" lies in the huge amount of induced traffic that renders the solution unscalable [25, 38]. The problem of having a scalable P2P overlay network with no central control became a scientifically challenging problem and the efforts to solve it resulted in the emergence of what is known as "structured P2P overlay networks", referred to also by the term Distributed Hash Tables (DHTs).

The General Solution. The main approach that was introduced by the academics to build overlay networks was to let the set of cooperating peers act as a a distributed data structure with well-defined operations, namely a distributed hash table with the two primitive operations `Put(key,value)` and `Get(Key)`. The `Put` operation should result in the storage of the value at one of the peers such that any of the peers can perform the `Get`

operation and reach the peer that has the value. More importantly, both operations need to take a "small" number of hops. A first naive solution would be that every peer knows all other peers, and then every `Get` operation would be resolved in one hop. Apparently, that is not scalable. Therefore, a second constraint is needed. Each node should know a "small" number of other peers. From a graph-theory point of view, this means that a directed graph of a certain known "structure" rather than a random graph needs to be constructed with scalable sizes of both the outgoing degree of each node and the diameter of the graph.

1.2 Definitions and Assumptions

Values. The set of values $\mathcal{V}$ such as files, directory entries etc.. Each value has a corresponding key from the set $Keys(\mathcal{V})$. If a value is a file, the key could be, for instance, its checksum, a combination of owner, creation date and name or any such unique attribute.

Nodes. The set $\mathcal{P}$ of machines/processes also referred to as nodes or peers. $Keys(\mathcal{P})$ is the set of unique keys for members of $\mathcal{P}$, usually the IP addresses or public keys of the nodes.

The Identifier Space. A common and fundamental assumption of all DHTs is that the keys of the values and the keys of the nodes are mapped into one range using a hashing function. For instance, the IP addresses of the nodes and the checksums of files are hashed using SHA-1 [12] to obtain 128-bit identifiers. The term "identifier" is used to refer to hashed keys of items and of nodes. The term "identifier space" refers to the range of possible values of identifiers and its size is usually referred to by N. We use id as an abbreviation for identifier most of the time.

Items. When a new value is inserted in the hash table, its key is saved with it. We use the term "item" to refer to a key-value pair.

Equivalence of Nodes. The operations of adding a value, looking up a value, adding a new node (join), removing an existing node (leave) are all possible through any node $p \in \mathcal{P}$.

Autonomy of Nodes. The addition or removal of any node is a decision taken locally at that node and there is a distinction between graceful removals of nodes (leaves) and ungraceful removals (failures).

The first contact. Another fundamental assumption in all DHTs is that to join an existing set of peers who already formed an overlay network, a new peer must know some peer in that network. This knowledge in many systems is assumed to be acquired by some out-of-band method. Some systems discuss the possibility of obtaining the first contact through IP multicast, however, it is an orthogonal issue to the operation of any DHT.

Ambiguous terms. Since we are forced to use different terminology to refer to the same logical entities in different contexts, we try to resolve those ambiguities early by introducing the following equalities. Nodes = peer = contact= reference, overlay network = overlay graph, identifier=id, edge = pointer, "point to"= "be aware of" = "keep track of", routing table = outgoing edges, diameter = lookup path length, lookup = query. routing table size = outgoing arity. Also some times, letters like n, s, t, x are used to refer to nodes and values as well as their identifiers but the meaning should be clear from the context.

1.3 Comparison Criteria

The Overlay Graph. This is the main criteria that distinguishes systems from each other. For each overlay graph, we want to know how the graph looks like and what is the outgoing arity of each node in the graph.

Mapping Items Onto Nodes. For a given overlay graph, we want to know the relation between node ids and item ids, i.e. at which node should an item be stored?

The Lookup Process. A tightly coupled property with the overlay graph is how lookups are performed and what is the typical performance.

Joins, Leaves and Maintenance. How a new node is added to the graph and how a node is gracefully deleted from the graph? Joins and leaves make the graph change constantly and some maintenance process is usually required to cope with such changes, so how does this process take place and what is its cost?

Replication and Fault Tolerance. In addition to graceful removal of nodes, failures are usually harder to deal with. Replication is a tightly coupled property since it can be a technique to overcome failures effect or a method of improving efficiency.

Upper Services and Applications. When applicable, we enumerate some of the applications and services developed using a certain system.

Implementation. Since many systems are of a completely theoretical nature even for their services and applications, we try to give and idea about any available implementations of a system.

1.4 DHT Systems

1.4.1 Chord

The Overlay Graph. Chord [42, 43] assumes a circular identifier space of size N. A Chord node with identifier u has a pointer to the first node following it clockwise on the identifier space ($Succ(u)$) as well as the first node preceding it ($Pred(u)$). The nodes therefore form a doubly linked list. In addition to those, a node keeps $M = \log_2(N)$ pointers called fingers. The set of fingers of node u is $F_u = \{(u, Succ(u + 2^{i-1}))\}, 1 \leq i \leq M$, where the arithmetic is modulo N;

The intuition of that choice of edges is that a node perceives the circular identifier space as if it starts from its id. The edges are, then, chosen such as to be able to partition the space into two halves, partition one of the halves into two quarters, and so forth.

In Figure 1.1(a), we show a network with an id space $N = 16$. Each node has $M = \log_2(N) = 4$ edges. The network contains nodes with ids $0, 3, 5, 9, 11, 12$. The general policy for constructing routing tables is shown in figure 1.1(b). Node n chooses its pointers by positioning itself at the start of the identifier space. It chooses to have the pointers to the

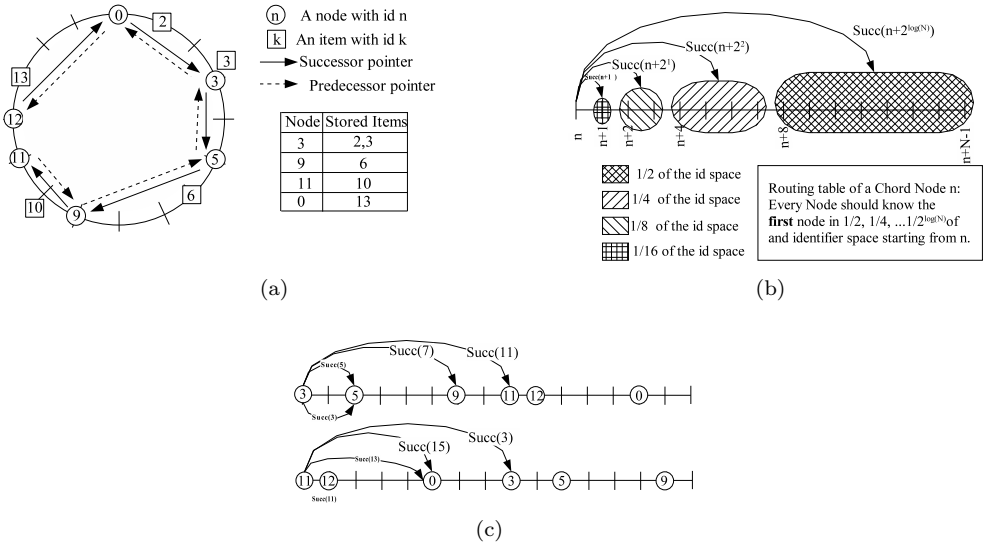


Figure 1.1: (a) A chord network with $N = 16$ populated with 6 nodes and 5 items. (b) The general policy for Chord's routing tables. (c) Example routing tables for nodes 3 and 11.

successors of the ids $n + 2^0$, $n + 2^1$, $n + 2^2$, and $n + 2^3$. The last pointer $n + 2^3$, divides the space into two halves. The one before it $n + 2^2$ divides the first half into two quarters and so forth. However, there may not exist a node at the desired position so its successor is taken instead. Figure 1.1(c) shows the routing entries of node 3 and 11.

Mapping Items Onto Nodes. As shown in figure 1.1(a), an item is stored at the first node that follows clockwise on the circular identifier space. If items with ids 2, 3, 6, 10,13 are to be stored in the network given above, then {2,3} will be stored at 3; {6} at 9; {10} at 11; and {13} at 0.

The Lookup Process. The lookup process comes as a natural result of how the id space is partitioned. Both the insertion and querying of items depend on finding the successor of an id. For example, assume that node 11 wants to insert a new item with id 8, the lookup is forwarded to node 3, which is the closest preceding finger - from the point of view of 11 - to the id 8. Node 3 will act similarly and forward the query to node 5 because 5 is the closest preceding finger for 8 from the point of view of 5. Node 5 finds that 8 is between itself and its successor 9. And therefore, returns 9 as an answer to the query through the reverse path[1]. In all cases, upon getting the answer, node 11's application layer should contact node 9's application layer and ask for the storage of some value under the key 8. Any node looking for the key 8 can act similarly and in no more than M hops[2], a node will discover the node at which 8 is stored. In general, under normal conditions a lookup takes $O(\log_2(N))$ hops.

Joins, Leaves and Maintenance. To join the network, a node n performs a lookup for its own id through some first contact in the network and inserts itself in the ring between its

[1]This is known as the recursive method. Another suggested approach in the Chord papers is an iterative method where all the answers path by the node at which the lookup originated, i.e. instead of the path being $11 \rightarrow 3 \rightarrow 5 \rightarrow 3 \rightarrow 11$, in an iterative lookup the path will be $11 \rightarrow 3 \rightarrow 11 \rightarrow 5 \rightarrow 11$. A third approach adopted in other systems like e.g. [2] would be to continue to the destination and send the result to the origin of the lookup, i.e. $11 \rightarrow 3 \rightarrow 5 \rightarrow 9 \rightarrow 11$.

[2]Chord counts a remote procedure call and the response to it as one hop.

successor s and the predecessor of s using a periodic stabilization algorithm. Initialization of n's routing table is done by copying the routing table of s or letting s lookup each required edge of n. The subset of nodes that need to adjust their tables to reflect the presence of n, will eventually do that because all nodes run a stabilization algorithm that periodically goes through the routing table and looks up the value of each edge. The last task is transfer part of the items stored at s, namely items with id less than or equal to n need to be transferred to n and that is also handled by the application layers of n and s.

Graceful removals (leaves) are done by first transferring all items to the successor and informing the predecessor and successor. The rest of the fingers are corrected by the virtue of the stabilization algorithm.

Replication and Fault Tolerance. Ungraceful failures have two negative effects. First, ungraceful failures of nodes cause loss of items. Second, part of the ring is disconnected leading to the inability of looking up certain identifiers. Let alone if a set of adjacent nodes fail simultaneously. Chord tackles this problem by letting each node keep a list of the $\log_2(N)$ nodes that follow it on the circle. The list serves two purposes. First, if a node detects that its successor is dead, it replaces it with the next entry in its successor list. Second, all the items stored at a certain node are also replicated on the nodes in the successor list. For an item to be lost or the ring to be disconnected, $\log_2(N) + 1$ successive nodes have to fail simultaneously.

Upper Services and Applications. A couple of applications such as a cooperative file-system [9], a read/write file system [29] and a DNS directory [8] were built on top of chord. As a general purpose service, a broadcast algorithm was also developed for Chord [10].

Implementation. The main implementation of Chord is that by its authors in C++ at [44] where a C++ discrete-event simulator is also available. Naanou [19] is a C# implementation of Chord with a file-sharing application on top of it.

1.4.2 Pastry

The Overlay Graph. The overlay graph design of Pastry [39] in addition to aiming to achieving logarithmic diameter with a logarithmic node state, also tries to target the issue of locality. In general, as a result of obtaining the node ids by hashing IP numbers/Public Keys, nodes with adjacent node ids may be farther apart geographically. Differently said, two machines in one country, would communicate through a machine in another continent just because the hash of their ids will be far apart in the id space.

Pastry assumes a circular identifier space and each node has a list containing $\frac{L}{2}$ successors and $\frac{L}{2}$ predecessors known as the leaf set. A node also keeps track of M nodes that are close according to another metric other than the id space like, for instance, network delay. This set is known as the neighborhood set and is not used during routing but used for maintaining locality properties. The third type of node state is the main routing table. It contains $\lceil \log_{2^b}(N) \rceil$ rows and $2^b - 1$ columns. L, M and b are system parameters.

Node ids are represented as string of digits of base 2^b. In the first row, the routing table of a node contains node ids that have a distinct first digit. Since the digits are of base 2^b, a

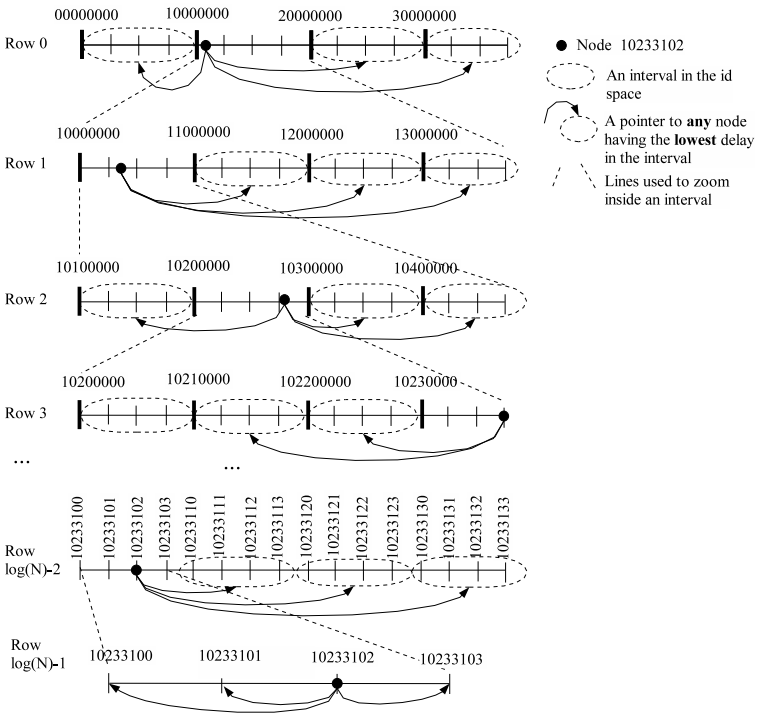


Figure 1.2: Illustration of how the Pastry node 10233102 **chooses its routing edges in an identifier space of size** $N = 2^{128}$ **and encoding base** $2^b = 4$**.**

node needs to know $2^b - 1$ nodes for each possible digit except its own.

The second row of a node with id n contains $2^b - 1$ nodes that share the first digit with n but differ in the second digit. The third row contains nodes that share the first and second digit of n but differ in the third and so forth. We stress that -1 in $2^b - 1$ is because in each row the node itself would be the best match for one of the columns, therefore we do not need to keep an address of it. Figure 1.2 illustrates how the the id space is partitioned using this prefix matching scheme.

As one can observe, for each of the constraints about the node ids contained in a routing table, there exists many satisfying nodes. Therefore the node with the lowest network delay or the best according to some other criteria is included in the routing table.

Mapping Items Onto Nodes. An item in Pastry is stored at the node that is numerically closest to the id of the item. Such a node will have the longest matching prefix.

The Lookup Process. To locate the closest node to an id x, a node n checks first if x falls within the range of node ids covered by its leaf set. If so, it is forwarded to such node. Otherwise, the lookup is forwarded to the node in the interval that x belongs to, that is to a node that shares more digits than the shared prefix between n and x. If no such node is found in n's routing table, the lookup is forwarded to the numerically closest node to x. The later case does not happen so often provided that the ids are uniformly distributed. With the matching of one digit of the sought id in each hop, after $\log_{2^b}(N)$ hops a lookup is resolved.

Joins, Leaves and Maintenance. When a node n joins the network through a node t, then t is usually in the proximity of n and thus the neighborhood set of t is suitable for n. Due to the construction of the routing tables in Pastry, n performs a lookup for its own id to figure out the numerically closest node s to n. It can take the i^{th} row from the i^{th} node on the path from t to s and use those rows in initializing its routing table. Moreover, the leaf set of s is a good initialization for the leaf set of n. Finally, n informs every node in its neighborhood set, leaf set and routing table of its presence. The cost is about $3 \times 2^b \log_{2^b} N$.

Node departures are detected as failures and repaired in a routing table by asking a node in the same row of the failed node for its entry on the failed position.

Replication and Fault Tolerance. Pastry replicates an item on the k closest nodes in its leaf set. This serves in saving an item after a node loss and in the mean time, the replicas act as cached copies that can contribute in finding an item more quickly.

Upper Services and Applications. A number of applications and services were developed on top of Pastry such as, SCRIBE [7] for multicasting and broadcasting. PAST [40], an archival storage system. SQUIRREL [20], a co-operative web caching system. SplitStream [6], a high-bandwidth content distribution .

Implementation. FreePastry [14] is an open-source Java implementation of the Pastry system.

1.4.3 Tapestry

Tapestry [46] is one of the earliest and largest efforts on structured P2P overlay networks. Like Pastry, it is based on the earlier work of a Plaxton [35] mesh. We will not describe the details of Tapestry due to the large similarity with Pastry. However, we have to point out that as a software, it is probably one of the most mature implementations of a structured overlay network. In addition to network simulation, Tapestry has been evaluated using a more realistic environment, namely PlanetLab [34], a globally distributed platform with machines all over the world that is used for testing large-scale systems.

Tapestry is a corner-stone project in the larger Oceanstore [22] project for global-scale persistent storage. Other applications based on Tapestry include the steganographic file system Mnemosyne [18], Bayeux [48] an efficient self-organizing application-level multicast system, and SpamWatch [47] a decentralized spam-filtering system.

1.4.4 Kademlia

The Overlay Graph. The Kademlia [28] graph partitions the identifier space exactly like Pastry. However, it is presented in a different way where node ids are leafs of a binary tree with each node's position is determined by the shortest unique prefix of its id. Each node divides the binary tree into a series of successively lower subtrees that don't contain the node id and keeps at least one contact in each of those subtrees. For instance, a node with id 3 has the binary representation 0011 in an identifier space of size $N = 16$. Since its prefix of length 1 is the digit 0 then it needs to know a node whose first digit is 1. Since its prefix

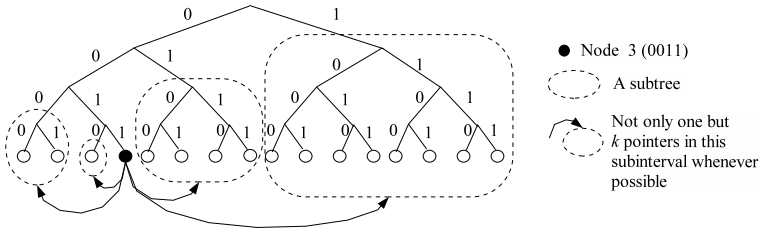


Figure 1.3: The pointers of node 3 (0011) in Kademlia. The same partitioning of the identifier space as in Pastry with binary-encoded digits.

of length 2 is 00, then it needs to know a node with prefix 01. Since its prefix of length 3 is 001, then it needs to know a node with prefix 000. Finally, since its prefix of length four is 0011, then it needs to know a node with a prefix 0010. This policy is illustrated in figure 1.3 which results in a space division exactly like Pastry with the special case of a binary encoding of the digits.

Kademlia does not keep a list of nodes close in the identifier space like the leaf set or the successor list in Chord. However, for every subtree/interval in the identifier space it keeps k contacts rather than one contact if possible, and calls a group of no more than k contacts in a subtree a k-bucket.

Mapping Items Onto Nodes. Kademlia defines the notion of distance between two identifiers to be the value of the bitwise exclusive or (XOR) of the two identifiers. An item is stored at the node whose XOR difference between the node id and the item id is minimal.

The Lookup Process. To increase robustness and decrease response time, Kademlia performs lookups in a concurrent and iterative manner. When a node looks up an id, it checks to which subtree does the id belong and forwards the query to α randomly selected nodes from the k-bucket of that subtree. Each node possibly returns back a k-bucket of a smaller subtree closer to the id. From the returned bucket, another α randomly selected nodes are contacted and the process is repeated until the id is found. When an item is inserted, it is also stored at the k closet nodes to its id. Because of the prefix matching scheme, similar to Pastry, a lookup is also resolved in $O(\log(N))$ hops.

Joins, Leaves and Maintenance. A new node finds the closest node to it through any initial contact and uses it to fill its routing table by querying about nodes in different subtrees. If it happens that a k-bucket is filled due to exposure to lots of nodes in a particular subtree, a least-recently-used replacement policy is applied. However, Kademlia makes use of statistics taken from existing peer-to-peer measurements studies which indicate that a node which stayed for a longer time in the past will probably stay connected longer in the future. Therefore, Kademlia can discard the knowledge of new nodes if it knew many other stable nodes in a given subtree.

Maintenance of the routing tables after joins and leaves depends on a technique that is different from the stabilization in Chord or the deterministic update of Pastry. Kademlia maintains the routing tables by using the lookup traffic. The XOR metric results in every node receiving queries from the nodes contained in its routing table (Which is not the case in a system like Chord). Consequently, the reception of any message from a certain node in a certain subtree is essentially an update of the k-bucket for that subtree. This approach

clearly minimizes the maintenance cost. However, it is not deeply analyzed.

Another maintenance task is that upon receiving multiple queries from the same subtree, Kademlia updates the latencies of the nodes in a particular k-bucket. This improves the choice of the nodes used for doing lookups and one could say that by doing that, Kademlia also takes into consideration network delay and locality.

Replication and Fault Tolerance. Since leaves are not deeply discussed, we assume that they are treated as failures. Kademlia fault tolerance depends mainly on the strong connectivity since it keeps k contacts per subtrees and not only one and this makes the probability of a disconnected graph low.

Also as mentioned above, Kademlia stores k copies of an item on the k closest nodes to its id. The nodes are also republished periodically. The policy for republishing is that any node that sees itself closer to an item id than all the nodes it knows about, gives it to $k-1$ other nodes.

Applications and Implementation. Kademlia is probably the one DHT that got a relatively wider non-academic adoption by being used in two file-sharing applications, namely Overnet [32] and Emule [11].

1.4.5 HyperCup

While it has been mentioned many times in the literature that systems like Chord and Pastry, for instance, are approximations of Hypercubes, those works were not presented that way by their authors. HyperCup [41] is a system that presents a way to construct and maintain Hypercubes in a dynamic setting. The performance of HyperCup is similar to the many other DHTs with logarithmic order for both the routing table size and the lookup path length under particular uniformity assumptions. HyperCup also defines a broadcast algorithm based on the concept of a spanning tree of all nodes. A distinguished feature of HyperCup is that it addresses semantic search based on ontological terms. Nodes with similar ontologies are clustered together such that a search by a certain ontological term is achieved as a localized broadcast within a cluster.

1.4.6 DKS

The Overlay Graph. DKS [2] could be perceived as an optimal generalization of Chord to provide shorter diameter with larger routing tables. In the mean time, DKS could be perceived as a meta-system from which other systems could be instantiated. DKS stands for Distributed k-ary Search and it was designed after perceiving that many DHT systems are instances of a form of k-ary search. Figure 1.4 shows the division of the space done in DKS. You can see that it has in common with Chord that each node perceives itself as the start of the space. In the mean time, like Pastry each interval is divided into k rather than 2 intervals.

Mapping Items Onto Nodes. Along with the goal of DKS to act as a meta-system, mapping items onto nodes is also left as a design choice. A Chord like mapping is a valid as

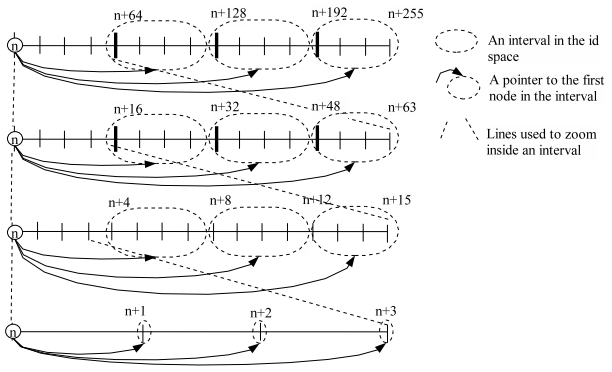


Figure 1.4: Illustration of how a DKS node divides the space in an identifier space of size $N = 2^8 = 256$.

a simple first choice. However, different mappings are possible as well.

The Lookup Process. A query arriving at a node is forwarded to the first node in the interval to which the id of the node belongs. Therefore, a lookup is resolved in $\log_k(N)$ hops.

Joins, Leaves and Maintenance. Unlike Chord, DKS avoids any kind of periodic stabilization both for the maintenance of the successors, the predecessor and the routing table. Instead, it relies on three principles, local atomic actions, correction-on-use and correction-on-change. When a node joins, a form of an atomic distributed transaction is performed to insert it on the ring. Routing tables are then maintained using the correction-on-use technique, an approach introduced in DKS. Every lookup message contains information about the position of the receiver in the routing table of the sender. Upon receiving that information, the receiver can judge whether the sender has an updated routing table. If correct, the receiver continues the lookup, otherwise the receiver notifies the sender of the corruption of his routing table and advises him about a better candidate for the lookup according to the receiver's knowledge. The sender then contacts the candidate and the process is repeated until the correct node for the routing table of the sender is used for the lookup.

By applying the correction-on-use technique, a routing table entry is not corrected until there is a need to use it in some lookup. This approach reduces the maintenance cost significantly. However, the number of joins and leaves are assumed to be reasonably less than the number of lookup messages. In cases where this assumption does not hold, DKS combines it with the correction-on-change technique [4]. Correction-on-change notifies all nodes that need to be updated upon the occurrence of a join, leave or failure.

Replication and Fault Tolerance. In early versions of DKS, fault tolerance was handled similar to Chord where replicas of an item are placed on the successor pointers. In later developments [16], DKS tries to address replication more on the DHT level rather than delegating most of the work to the application layer. Additionally, to avoid congestion in a particular segment of the ring, replicas are placed in dispersed well-chosen positions and not on the successor list. In general, for the correction-on-use technique to work, an invariant is maintained where the predecessor pointer has always to be correct and that is provided by the atomic actions on the circle.

Upper Services and Applications. General purpose broadcast [15] and multicast [3]

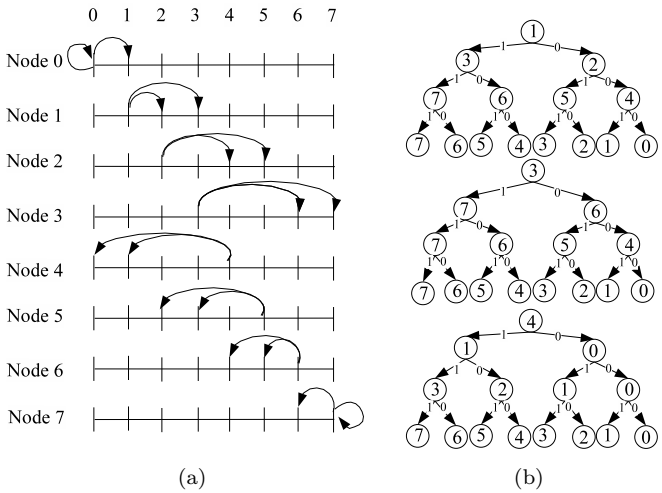


Figure 1.5: (a) The pointers of all the nodes in a complete Koorde network where $N = 8$. Every node n points to nodes of ids $2n$ and $2n + 1$. (b) Examples of how nodes 1, 3 and 4 reach other nodes by matching the destination id digit by digit starting from the most significant bit.

algorithms were developed for DKS.

1.4.7 P-Grid

P-Grid [1] is a system based on randomized algorithms assuming that there will be random interactions between nodes of the overlay. In the mean time, P-Grid ensures that those peers are arranged in a graph most similar to the overlay graph of Pastry. A unique assumption of P-Grid is that nodes do not have constant ids, instead, ids change over time in order for the identifier space to be partitioned fairly among them. The property is not only used for decreasing the lookup path length but also for balancing items among nodes. A file-sharing application with the same name is implemented in Java and available at [33]

1.4.8 Koorde

The Overlay Graph. Koorde [21] is based on the DeBruijn graph [27]. Koorde stresses the point that a constant number of outgoing edges per node is enough for having a logarithmic lookup length. The DeBruijn graph is an example capable of doing that. The significance of a constant number of edges is that the maintenance overhead is lower compared to a logarithmic number as is the case in all the previous DHTs we have shown so far. In figure 1.5(a) we show the pointers of all the nodes of a Koorde graph of eight nodes. A node with id n has edges to nodes $2n$ and $2n + 1$ in a circular identifier space like Chord. We denote the first and the second edge of node n $E_n \circ 0$ and $E_n \circ 1$ respectively.

Mapping Items Onto Nodes. Exactly like Chord.

The Lookup Process. When a node n needs to lookup an id x represented as a string of binary digits $d_1 d_2 ... d_{\log_2(N)}$, it takes the top bit d_1, if it is a 0, it forwards the query to $E_n \circ 0$ otherwise to $E_n \circ 1$. The second node looks at the remaining string $d_2 ... d_{\log_2(N)}$ and acts similarly. After, at most, $\log_2(N)$ hops a query is resolved. Figure 1.5(b) shows what paths nodes $1, 3$ and 4 take to reach any node in the network. The Koorde paper also elaborates on an algorithm to handle networks where not all the nodes are present in the id space. Each node tries to locally traverse imaginary hops for nodes that do not exist.

Joins, Leaves and Maintenance. Exactly like Chord. In fact, the authors say that Koorde could be perceived as a Chord system with a constant instead of a logarithmic number of fingers. Stabilization is also the basic mechanism for maintenance.

Replication and Fault Tolerance. For fault tolerance to be realized, an out-degree less than $\log(N)$ nodes has to be maintained, otherwise a node will loose all its contacts very easily. This makes the advantage of a constant node state invalid. However, since with k edges, Koorde provides $\log_k(N)$ diameter. Then with $\log_k(N)$ edges, it provides $\log_{\log_k(N)}(N) = \frac{\log(N)}{\log(\log(N))}$ diameter, which is an advantage over the logarithmic class of DHTs.

Load Balancing. The load balancing of items onto nodes will depend on the uniform distribution exactly like Chord. However, another load-balancing issue arises which is the load of message passing on each node. In a DeBruijn graph, some nodes will have more traffic than others by a factor of $O(\log(N))$ of the average traffic load. For example, in the network illustrated in figure 1.5, if every node would send a message to every other node in the network, not all the nodes will endure the same number of messages; 12 messages will be routed via a node like 7 while 21 messages will be routed via a node like 3.

1.4.9 Distance Halving

The Overlay Graph. The Distance Halving (DH[3]) [30] distributed hash table is another system based on the DeBruijn graph like Koorde. However, the way of building the graph is somewhat different. The DH is based on an approach called the continuous-discrete approach for building graphs. To build a DeBruijn graph with this approach, the identifier space is normalized into a continuous space represented by the interval $[0, 1[$. Nodes are points in that interval. Each node y has two edges, a left edge and a right edge denoted $\ell(y)$ and $r(y)$ respectively where $\ell(y) = \frac{y}{2}$ and $r(y) = \frac{y}{2} + \frac{1}{2}$. Given the set of points and their edges, a discretization step is done to build the graph. The set of points are denoted by $\vec{x}$. The points of $\vec{x}$ divide the space into n segments. The segment of a point x_i, $S(x_i) = [x_i, x_{i+1})$, $(i = 1...n-1)$ and $S(x_n) = [x_{n-1}, 1) \cup [0, x_1)$. If a node y has an edge that belongs to the segment of some node z, then there is an edge in the discrete graph between y and z. One can also notice that the segments are defined in a way that realizes a circular identifier space.

The intuition behind that graph is that every node divides the space into two intervals and keeps a pointer to a node that is in the middle of the left interval and a pointer in the middle of the right interval. Figure 1.6(a) shows the pointers of all the nodes in a DH network of size $N = 8$. Figure 1.6(b) shows the paths to all possible destinations starting from node 1.

[3]Please do not confuse this abbreviation with the abbreviation of a Distributed Hash Table (DHT).

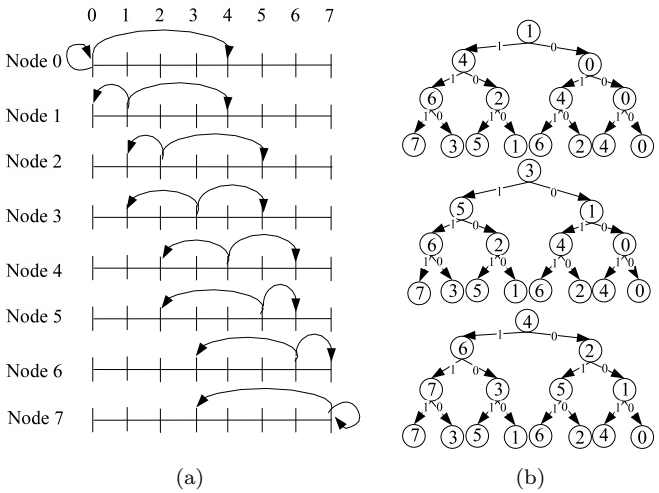


Figure 1.6: (a) The pointers of all the nodes in a complete Distance-Halving network where $N = 8$. (b) Examples of how nodes 1 reaches other nodes by matching the destination id digit by digit starting from the least significant bit.

Mapping Items Onto Nodes. Exactly like Chord, Koorde. In DH terminology, an item is stored at node y where the id of the item belongs to $S(y)$.

The Lookup Process. The lookup process, similar to many other DHTs, is done by the prefix matching of the sought id digit by digit. The lookup is forwarded to the node pointed to by the left edge for matching a 0 digit and to the right edge for matching a 1 digit. The lookup path length is thus $O(\log_2(N))$.

Joins, Leaves and Maintenance. A new node n joins a DH network by looking up the node s such that n belongs to $S(s)$. n then uses s to lookup its left and right edges. By the construction of DH, a node can easily know the nodes that are pointing to it. Therefore a node can easily compute the nodes that needs to be updated and notifies them of n's existence. Updating of others upon a leave is done in the same way. The transfer of the items upon a join or a leave is also similar to Chord.

Replication and Fault Tolerance. DH recognizes the problem of failures that can lead to a disconnected graph and advocates an additional state of $O(\log(N))$ pointers. That comes in agreement with Koorde's reasoning and emphasizes that the main advantage of having a constant degree graph will be compromised if fault tolerance is to be considered. However, with a logarithmic degree, those types of graphs can offer a diameter of $\frac{\log(N)}{\log(\log(N))}$.

Other Comments. The formal analysis of the DH graph and the continuous discrete approach are both useful tools that help gaining more understanding of the properties of a DHT system. The discussion of the smoothness of the graph which is a term used by the authors to quantify the uniformity of distribution of the ids is quite unique. It was noted in other DHT systems that uniform distribution could affect the performance but in DH, an analysis of the magnitude of that effect is provided.

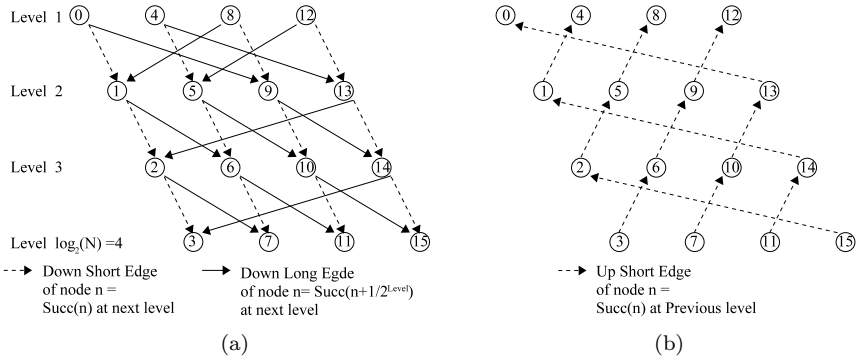


Figure 1.7: The Butterfly edges of a complete Viceroy network with $N = 16$ nodes. (a) The down edges. (b) The up edges.

1.4.10 Viceroy

The Overlay Graph. Viceroy [24] is based on the Butterfly [26] network. Like many other systems, it organizes nodes into a circular identifier space and each node has successor and predecessor pointers. Moreover, nodes are arranged in $\log_2(N)$ levels numbered from 1 to $\log_2(N)$. Each node apart from nodes at level 1 have an "up" pointer and every node apart from the nodes at the last level have 2 "down" pointers. There is one short and one long "down" pointers. Those three pointers are called the Butterfly pointers. All nodes also have pointers to successors and predecessors pointers on the same level. In that way, each node has a total of 7 outgoing pointers.

Figure 1.7(a) shows the down pointers of a network of $N = 16$ nodes where all nodes are present. Figure 1.7(b) shows the up pointers of all nodes. For simplicity, the successor pointers of the ring and the levels are not illustrated[4].

Mapping Items Onto Nodes. Exactly like Chord.

The Lookup Process. To lookup an item x, a node n follows its up pointer until it reaches level 1. From there, it starts going down using the down links. In each hop, it should traverse a pointer that does not exceed the target x. For example, if node 1 is looking up the id 10, first it will follow its up pointer and reach 4 which is at level 1. At node 4 there are two choices either to use the short pointer to 5 or the long pointer to 13, since 5 precedes the target 10, the pointer to 5 is followed. At node 5, there is a direct pointer to 10. In another example, for reaching id 15 from node 3, the path will be $3 \stackrel{up}{\rightarrow} 6 \stackrel{up}{\rightarrow} 9 \stackrel{up}{\rightarrow} 12 \stackrel{down}{\rightarrow} 13 \stackrel{down}{\rightarrow} 14 \stackrel{down}{\rightarrow} 15$. From the last example, we can see that in a worst case, we can traverse all the levels up and down, i.e. $2 \times log(N)$ hops. Needless, to say that the example includes a simplified network where all the nodes are present. When the graph is sparse, the reasoning is slightly more complicated, however the expected lookup path length is still $O(\log(N))$.

Joins, Leaves and Maintenance. To join, a node looks up its successor s, fixes the ring pointers and takes the required items from s. After that, it selects a level based on the estimation of the number of nodes. It finds, by a combination of lookups and stepping

[4]In fact, some of them coincide with the Butterfly pointers.

on the ring, the rest of the pointers (successor and predecessor at the selected level, up and down pointers).

To leave, a node disconnects all its pointers, the concerned nodes consequently are aware and lookup for replacements. Additionally, the stored items are transferred to the successor.

Replication and Fault Tolerance. Viceroy does not deeply discuss ungraceful failures nor replication but refers to Lynch et al.'s paper [23] for a general approach in handling failures in DHTs.

Implementation. There exists a Java implementation of Viceroy at [45]. This homepage includes also a visualization applet that can illustrate the main topology, lookups, joins and leaves in Viceroy.

Other Comments. While the intuitive analysis might lead to thinking that nodes at higher levels endure more lookup traffic, Viceroy's analysis shows that the congestion is not that bad, however such a proof is beyond the scope of that chapter.

1.4.11 Ulysses

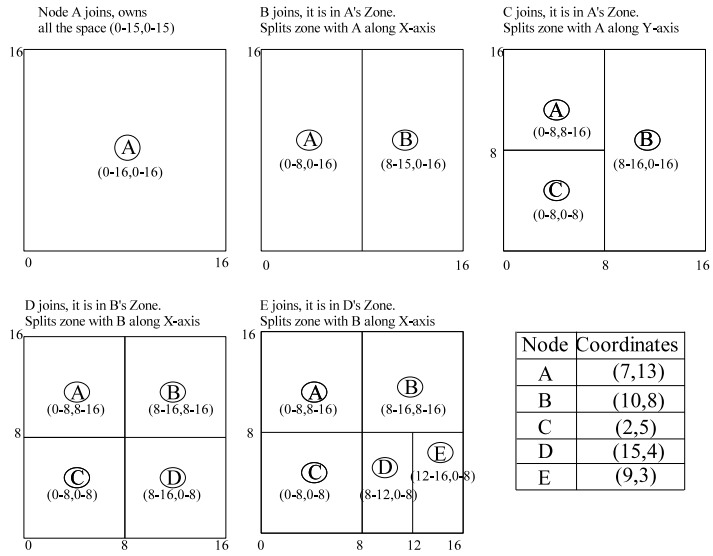
Ulysses is another system based on the Butterfly graph. It achieves the known limits of routing table and lookup path length, $O(\log(N))$ and $\frac{\log(N)}{\log(\log(N))}$ while accounting for joins, leaves and failures. In that sense, it agrees with the conclusions of Koorde, Distance-Halving and Viceroy and shows a second way of building a Butterfly network. Ulysses also depends on periodic stabilization for maintenance of the graph. Like Distance-Halving, it discusses the elimination of congestion. Ulysses also has an interesting discussion on the optimization of the average lookup path length.

1.4.12 CAN

The Overlay Graph. CAN [36] is in a class of its own. The design of the graph is based on a d-dimensional coordinate space. Like all other systems, the nodes and items are mapped onto a virtual space using a uniform hashing function, but the hashing is applied d times to get the d coordinates. For instance, in a 2-dimensional discrete coordinate space, an IP address or key of a file would be hashed once to obtain an x value and another time to obtain a y value. The coordinate space is dynamically partitioned among all the nodes in the system such that every node"owns" its distinct zone within the overall space. Figure 1.8 shows a discrete coordinate space of 16×16 partitioned among 5 nodes.

Mapping Items Onto Nodes. An item with key k is stored at the node that owns the zone onto which k is mapped. Two nodes are neighbors, i.e. have pointers to each other if their zones have common sides.

The Lookup Process. A lookup is achieved by using the straight line path through the Cartesian space from source to destination.



Node	Coordinates
A	(7,13)
B	(10,8)
C	(2,5)
D	(15,4)
E	(9,3)

Figure 1.8: The process of 5 nodes joining a CAN network.

Joins, Leaves and Maintenance. A new node w joins by selecting a random point P, it sends to its initial contact in the network u a JOIN message containing P. Node u consequently routes the message to the node v that owns the zone in which P lies. The zone of v is then split between v and w. Zones are split along the x axis first then along the y axis. Upon a split, the new node learns its neighbors from the previous owner. Neighbors of a new node are neighbors of the previous owner plus the previous owner itself. The new node informs its neighbors of the change. The cost of join in that way is $O(d)$. Finally, items that belong to the new node are obtained from the previous owner.

The leave process is the reverse, a node informs its neighbors of its leaving and merges its zone with a neighbor to produce a valid zone. If no valid zone could be formed, the items are transferred to a neighbor owning the smallest zone.

Under normal conditions, a node sends periodic updates to each of its neighbors given them its zone coordinates. Additionally, there is a background zone-balancing process that tries to reconfigure zones after a series of joins and leaves.

Replication and Fault Tolerance. There are two ways of detecting failures in CAN, the first if a node tries to communicate with a neighbor and fails, it takes over that neighbor's zone. The second way of detecting a failure is by not receiving the periodic update message after a long time. In the second case, the failure would probably be detected by all the neighbors, and all of them would try to take over the zone of the failed node, to resolve this, all nodes send to all other neighbors the size of their zone, and the node with the smallest zone takes over.

Replication in CAN is achieved in two ways. The first way is to use α hashing functions to map an item to α points. When retrieving an item, α queries are sent and α responses are received. The second way is to create multiple instances of the coordinate space. Each instance is called a "reality". If a node storing an item is dead in one reality, the item can be retrieved from one of the other realities because the item would be stored at other nodes in the other realities.

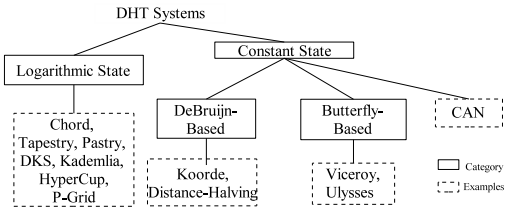


Figure 1.9: A classification of DHT systems based on the size of the node state and underlying graph.

Category	Node State	Lookup Path Length
Logarithmic sate	$O(\log(N))$	$O(\log(N))$
DeBruijn & Butterfly (per se)	$O(k)$	$O(\log(N))$
DeBruijn & Butterfly ($k = O(\log(N))$)	$O(\log(N))$	$\frac{O(\log(N))}{O(\log(\log(N)))}$
CAN (per se)	$O(k)$	$O(kN^{1/k}))$
CAN ($k = O(\log(N))$)	$O(\log(N))$	$O(\log(N)N^{1/\log(N)})) = O(\log(N))^a$

[a] Since $N^{1/\log(N)}$ is a constant factor.

Table 1.1: Summary of Node State and Lookup Path Length for the different categories of systems.

Latency. Every node in CAN keeps round-trip-time (RTT) of its neighbors. When selecting a path for a lookup, a CAN node forwards to the neighbor with maximum ratio of progress to RTT. CAN also has a mechanism for nodes to choose their points so as to make points near in the IP network also near in the Cartesian space, the technique uses root DNS servers as landmarks from which a node can approximate to which other nodes it is near in the IP network.

Upper Services. A multicast protocol is available for CAN [37]. Some work has also been done on richer queries such as range queries in [5].

1.5 Summary

1.5.1 The Overlay Graph

We summarize the different overlay graphs by providing a classification based on the size of the node state as shown in figure 1.9. The first category is for systems that keep a logarithmic number of routing entries. Most DHT systems are in that category. A common property in that category is the logarithmic order lookup path length. The second category includes systems that use a constant number of routing entries. CAN is in a class of its own as it provides a polynomial order lookup path length. Other systems in the same category include the DeBruijn-based and the Butterfly-based DHTs and such systems offer a logarithmic path length. Naturally, one can instead set the constant of the constant-state systems to a value logarithmic in the number of nodes and get a shorter lookup length. Table 1.1 summarizes those performance trade-offs.

1.5.2 Mapping Items Onto Nodes

Four ways of assigning items to ids are identified and summarized in table 1.2.

Assignment policy	Example Systems
Item assigned to successor on the ring	Chord, DKS, Koorde, Viceroy, DH, Ulysses
Item assigned to numerically closet node	Pastry, Tapestry
Item assigned to XOR closest node	Kademlia
Item assigned to zone owner	CAN

Table 1.2: The different policies for mapping items onto nodes.

In all those scenarios, the fair (load-balanced) assignment of items onto nodes relies on the uniform distribution of the hashing function. This is apart from P-Grid, where the network is in constant trial to load balance the items between nodes irrespective of the distribution of identifiers.

1.5.3 The Lookup Process

The lookup process is a direct result of the node state. Increasing more node decreases the lookup path length but increases the maintenance cost.

In some systems like Pastry, Tapestry, Kademlia and CAN, overlay hops are not the sole optimized metric, additionally network latency is addressed.

Congestion is a tricky issue related to the lookup process. Not only the lookup path length should be optimized, but it should not be the case that some nodes endure more traffic than others which is the case in the DeBruijn and Butterfly graphs. However, authors of systems that suffer from congestion try to adapt those graphs to eliminate congestion.

1.5.4 Joins, Leaves and Maintenance

Joins and leaves jeopardize the desired properties of any good graph, different systems have adopted different techniques to bring back the overlay graph to its ideal state. Table 1.3 enumerates those techniques.

Maintenance policy	Example Systems
Stabilization	Chord, Koorde, Viceroy, CAN
Use of Traffic	Kademlia
Determinism+ Stabilization	Pastry, Tapestry
Correction on-use + Correction on-change	DKS
Lazy+Randomized	P-Grid

Table 1.3: The different policies overlay graph maintenance policies.

Stabilization is the most common technique where routing table entries are periodically looked up and corrected. The use of traffic is adopted in Kademlia where the graph structure makes a node receive lookups from the same nodes it is pointing to. Pastry also depends on the structure of the graph where a new node can inform all the other nodes that need to be informed about it. Periodic activity is still needed though for collecting latency information. The correction-on-use introduced in DKS, relies on the presence of traffic as well but a receiving node can correct a sending node and no periodic activity is used. Where not sufficient alone, correction-on-use is complemented with a more deterministic technique, namely, correction-on-change. P-Grid has a unique correction mechanism, where the random interaction between peers can lead to the change of their ids in a way that causes eventual optimality of the graph.

1.5.5 Replication and Fault Tolerance

Replication is an essential tool for recovering items stored at failed nodes. Choice of nodes for replication is tightly coupled with the policy for mapping items to nodes. Local vicinity is mostly chosen, for example, the successors on the circle or the k numerically closest nodes.

Fault tolerance is one of the most challenging and open areas in structured overlay networks. Some systems can cope with the failure of a small number of nodes at a time. However, dealing with a large number of simultaneous failures is harder. A constant- state routing table is an advantage that has to be given up if a large number of simultaneous failures is to be tolerated. Nodes will have to keep their node state to at least logarithmic order to be able to cope with $N/2$ randomly-distributed nodes failing simultaneously or the failure of $O(\log(n))$ adjacent node ids simultaneously.

1.6 Open Problems and Other Issues

Some of the open issues related to the aspects we discussed in that chapter include the following: *Reducing maintenance cost.* While we have seen different techniques for dealing with the maintenance of the overlay structure, any optimization in that aspect is important for the overall performance of a DHT. *State-performance trade-off.* The trade-off between node-state and lookup path length is fundamental. The current known limit is that a constant node state can provide logarithmic path length but if fault-tolerance is to be addressed, more state is required. It is still an open question whether a constant state suffices for a fault-tolerant system while preserving the logarithmic path length and without introducing congestion. *Performance of existing DHTs.* At the time of writing of this chapter, while many systems have been introduced, less work has been dedicated to measuring their performance in different aspects and for different applications. *Heterogeneity.* Many of the desirable DHT properties depend on some kind of uniform distribution. In practice, aspects like connection time or bandwidth are evidently uniformly distributed for a given set of peers. Therefore, supporting heterogeneity is another open issue from a practical point of view. *Search and indexing.* All DHTs have the simple "put-get" interface. Consequently, the knowledge of the key of the sought item is assumed. However, an efficient listing of all items that have some common property is rather challenging in a DHT. *Grid computing.* Since both Grid computing and P2P computing are both forms of resource sharing, it is a current hot topic to investigate how results of both areas can be mutually beneficial. Other unrelated

active and challenging research topics include: Security, trust, anonymity, denial-of-service attacks, malicious node behavior, reputation and incentives.

1.7 Acknowledgments

We thank the members of the distributed systems lab at the Swedish Institute of Computer Science, Per Brand, Luc Onana Alima, Ali Ghodsi and Erik Aurell for the numerous discussions, insights, and the several points that they explained and clarified to us in the area of DHTs. This work was funded by the Swedish Vinnova PPC project and the European projects PEPITO and EVERGROW.

References

[1] Karl Aberer. P-Grid: A self-organizing access structure for p2p information systems. In *InProceedings of the Sixth International Conference on Cooperative Information Systems (CoopIS 2001)*, Trento, Italy, 2001.

[2] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. DKS(N; k; f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *The 3rd International Workshop On Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID 2003)*, Tokyo, Japan, May 2003. http://www.ccgrid.org/ccgrid2003.

[3] Luc Onana Alima, Ali Ghodsi, Per Brand, and Seif Haridi. Multicast in dks(n, k, f) overlay networks. In *The 7th International Conference on Principles of Distributed Systems (OPODIS'2003)*. Springer-Verlag, 2004.

[4] Luc Onana Alima, Ali Ghodsi, and Seif Haridi. A framework for structured peer-to-peer overlay networks. In *LNCS volume of the post-proceedings of the Global Computing 2004 workshop*. Springer-Verlag, 2004.

[5] Artur Andrzejak and Zhichen Xu. Scalable, Efficient Range Queires for Grid Information Services. In *2nd International Conference on Peer-To-Peer Computing*, pages 33–40, Linkping, Sweden, September 2002. IEEE Computer Scociety. ISBN-0-7695-1810-9.

[6] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[7] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[8] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving dns using chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[10] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient Broadcast in Structured P2P Netwoks. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[11] Emule. The emule file-sharing application homepage, 2004. http://www.emule-project.net/.

[12] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST., National Technical Information Service, Springfield, VA, April 1995.

[13] FreeNet, 2003. http://freenet.sourceforge.net.

[14] FreePastry. The freepastry homepage, 2004. http://www.cs.rice.edu / CS / Systems / Pastry / FreePastry.

[15] Ali Ghodsi, Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Self-correcting broadcast in distributed hash tables. In *In Series on Parallel and Distributed Computing and Systems (PDCS'2003)*, Calgary, 2003. ACTA Press.

[16] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. A novel replication scheme for load-balancing and increased security. Technical Report TR-2004-11, SICS, June 2004.

[17] Gnutella, 2003. http://www.gnutella.com.

[18] Steven Hand and Timothy Roscoe. Mnemosyne: Peer-to-peer steganographic storage. *Lecture Notes In Computer Science (IPTPS '02)*, pages 130–140, 2002.

[19] Clint Heyer. Naanou home page, 2004. http://naanou.sourceforge.net.

[20] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-

to-peer web cache. In *12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, ?-? 2002.

[21] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[22] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*. ACM, Nov 2000.

[23] Nancy Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in content addressable networks. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[24] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *InProceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, August 2002.

[25] E. P. Markatos. Tracing a Large-Scale Peer to Peer System: An Hour in the Life of Gnutella. In *The Second International Symposium on Cluster Computing and the Grid*, 2002. http://www.ccgrid.org/ccgrid2002.

[26] MathWorld. The butterfly graph, 2004. http:// mathworld.wolfram.com / Butterfly-Graph.html.

[27] MathWorld. The de bruijn graph, 2004. http:// mathworld.wolfram.com / deBruijn-Graph.html.

[28] Petar Maymounkov and David Mazires. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[29] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, December 2002. USENIX Association.

[30] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *InProceedings of SPAA 2003*, 2003.

[31] Napster. Open source napster server, 2002. http://opennap.sourceforge.net/.

[32] Overnet. The overnet file-sharing application homepage, 2004. http://www.overnet.com.

[33] P-Grid. The P-Grid homepage, 2004. http://www.p-grid.org.

[34] Planet-Lab. The planet-lab homepage. http://www.planet-lab.org.

[35] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[36] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proceedings of the ACM SIGCOMM '01 Conference*, Berkeley, CA, August 2001.

[37] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Third International Workshop on Networked Group Communication (NGC '01)*, 2001. http://www-mice.cs.ucl.ac.uk/ngc2001/.

[38] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping The Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems And Implications For System Design. *IEEE Internet Computing Journal*, 6(1), 2002.

[39] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 329-350 2001.

[40] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 188-201 2001.

[41] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. HyperCuP – hypercubes, ontologies and efficient search on peer-to-peer networks, May 2003. http://www-db.stanford.edu/ schloss/docs/HyperCuP-LNCS2530.ps.

[42] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[43] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, 2003.

[44] The Chord Project Home Page, 2003. http://www.pdos.lcs.mit.edu/chord/.

[45] Viceroy. Java implementation and visualization applet, 2004. http://www.ece.cmu.edu/ atalmy/viceroy.

[46] Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1):41–53, January 2004.

[47] Feng Zhou, Li Zhuang, Ben Y. Zhao, Ling Huang, Anthony D. Joseph, and John D. Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *Proc. of Middleware*, pages 1–20, Rio de Janeiro, Brazil, June 2003. ACM.

[48] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of NOSSDAV*, pages 11–20. ACM, June 2001.