8. Selected DHT Algorithms

Stefan Götz, Simon Rieche, Klaus Wehrle (University of Tübingen)

Several different approaches to realizing the basic principles of DHTs have emerged over the last few years. Although they rely on the same fundamental idea, there is a large diversity of methods for both organizing the identifier space and performing routing. The particular properties of each approach can thus be exploited by specific application scenarios and requirements.

This overview focuses on the three DHT systems that have received the most attention in the research community: Chord, Pastry, and Content Addressable Networks (CAN). Furthermore, the systems Symphony, Viceroy, and Kademlia are discussed because they exhibit interesting mechanisms and properties beyond those of the first three systems.

8.1 Chord

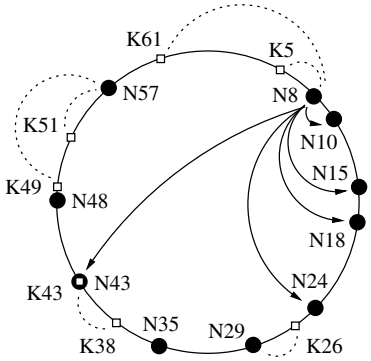
The elegance of the Chord algorithm, published by Stoica et al. [575] in 2001, derives from its simplicity. The keys of the DHT are l-bit identifiers, i.e., integers in the range $[0, 2^l - 1]$. They form a one-dimensional identifier circle modulo 2^l wrapping around from $2^l - 1$ to 0.

8.1.1 Identifier Space

Each data item and node is associated with an identifier. An identifier of a data item is referred to as a *key*, that of a node as an *ID*. Formally, the $(key, value)$ pair (k, v) is hosted by the node whose ID is greater than or equal to k. Such a node is called the *successor* of key k. Consequently, a node in a Chord circle with clockwise increasing IDs is responsible for all keys that precede it counter-clockwise.

Figure 8.1 illustrates an initialized identifier circle with $l = 6$, i.e., $2^6 = 64$ identifiers, ten nodes and seven data items. The successor of key $K5$, i.e., the node next to it clockwise, is node $N8$ where $K5$ is thus located. $K43$'s successor is $N43$ as their identifiers are equal. The circular structure modulo $2^6 = 64$ results in $K61$ being located on $N8$.

R. Steinmetz and K. Wehrle (Eds.): P2P Systems and Applications, LNCS 3485, pp. 95-117, 2005.
© Springer-Verlag Berlin Heidelberg 2005



Finger Table of N8		
Idx	Target ID	Successor
0	N8 + 1	N10
1	N8 + 2	N10
2	N8 + 4	N15
3	N8 + 8	N18
4	N8 + 16	N24
5	N8 + 32	N43

Fig. 8.1: A 6-bit Chord identifier space. Dotted lines indicate which nodes host which keys. Black lines represent the fingers of node $N8$.

8.1.2 Routing

Given a Chord identifier circle, all identifiers are well-ordered and keys and nodes are uniquely associated. Thus, each $(key, value)$ pair is located and managed on a single, well-defined node. The DHT is formed by the set of all $(key, value)$ pairs on all nodes of an identifier circle. The key to efficient lookup and modification operations on this data is to quickly locate the node responsible for a particular key.

For a very simple routing algorithm, only very little per-node state is required. Each node needs to store its successor node on the identifier circle. When a key is being looked up, each node forwards the query to its successor in the identifier circle. One of the nodes will determine that the key lies between itself and its successor. Thus, the key must be hosted by this successor. Consequently, the successor is communicated as the result of the query back to its originator.

This inefficient form of key location involves a number of messages linear to the number of nodes on the identifier circle. Chord utilizes additional per-node state for more scalable key lookups.

Each node maintains a routing table, the *finger table* (cf. Figure 8.1), pointing to other nodes on the identifier circle. Given a circle with l-bit identifiers, a finger table has a maximum of l entries. On node n, the table entry at row i identifies the first node that succeeds n by at least 2^{i-1}, i.e., $successor(n + 2^{i-1})$, where $1 \leq i \leq l$. In Figure 8.1, for example, the second finger of node $N8$ $(8+2^1 = 10)$ is node $N10$ and the third finger $(8+2^2 = 12)$ is node $N15$. The first finger of a node is always its immediate successor on the identifier circle.

As a finger table stores at most l entries, its size is independent of the number of keys or nodes forming the DHT. Each finger entry consists of a node ID, an IP address and port pair, and possibly some book-keeping

information. Even for large identifiers, e.g., $l = 256$, this is a relatively small amount of data per node which can be efficiently managed and searched. The routing information from finger tables provides information about nearby nodes and a coarse-grained view of long-distance links at intervals increasing by powers of two.

The Chord routing algorithm exploits the information stored in the finger table of each node. A node forwards queries for a key k to the closest predecessor of k on the identifier circle according to its finger table. When the query reaches a node n such that k lies between n and the successor of n on the identifier circle, node n reports its successor as the answer to the query.

Thus, for distant keys k, queries are routed over large distances on the identifier circle in a single hop. Furthermore, the closer the query gets to k, the more accurate the routing information of the intermediate nodes on the location of k becomes. Given the power-of-two intervals of finger IDs, each hop covers at least half of the remaining distance on the identifier circle between the current node and the target identifier. This results in an average of $O(log(N))$ routing hops for a Chord circle with N participating nodes. For example, a Chord network with 1000 nodes forwards queries, on average, in roughly $O(10)$ steps. In their experiments, Stoica et al. show that the average lookup requires $\frac{1}{2}log(N)$ steps.

8.1.3 Self-Organization

The Chord system described so far also needs to allow for nodes joining and leaving the system as well as to deal with node failures.

Node Arrivals

In order to join a Chord identifier circle, the new node first determines some identifier n. The original Chord protocol does not impose any restrictions on this choice. For example, n could be set at random assuming that the probability for collisions with existing node IDs is low in a identifier space large enough. There have been several proposals to restrict node IDs according to certain criteria, e.g., to exploit network locality or to avoid identity spoofing.

For the new node n, another node o must be known which already participates in the Chord system. By querying o for n's own ID, n retrieves its successor. It notifies its successor s of its presence leading to an update of the predecessor pointer of s to n. Node n then builds its finger by iteratively querying o for the successors of $n + 2^1$, $n + 2^2$, $n + 2^3$, etc. At this stage, n has a valid successor pointer and finger table. However, n does not show up in the routing information of other nodes. In particular, it is not known to its predecessor as its new successor since the lookup algorithm is not apt to determine a node's predecessor.

Stabilization Protocol

Chord introduces a *stabilization* protocol to validate and update successor pointers as nodes join and leave the system. Stabilization requires an additional predecessor pointer and is performed periodically on every node. The `stabilize()` function on a node k requests the successor of k to return its predecessor p. If p equals k, k and its successor agree on being each other's respective predecessor and successor. The fact that p lies between k and its successor indicates that p recently joined the identifier circle as k's successor. Thus, node k updates its successor pointer to p and notifies p of being its predecessor.

With the stabilization protocol, the new node n does not actively determine its predecessor. Instead, the predecessor itself has to detect and fix inconsistencies of successor and predecessor pointers using `stabilize()`. After node n has thus learnt of its predecessor, it copies all keys it is responsible for, i.e., keys between $predecessor(n)$ and n, while the predecessor of n releases them.

At this stage, all successor pointers are up to date and queries can be routed correctly, albeit slowly. Since the new node n is not present in the finger tables of other nodes, they forward queries to the predecessor of n even if n would be more suitable. Node n's predecessor then needs to forward the query to n via its successor pointer. Multiple concurrent node arrivals may lead to several linear forwardings via successor pointers.

The number of nodes whose finger table needs to be updated is in the order of $O(log(N))$ in a system with N nodes. Based on the layout of a finger table, a new node n can identify the nodes with outdated finger tables as $predecessor(n - 2^{i-1})$ for $1 < i \leq l$. However, the impact of outdated finger tables on lookup performance is small, and in the face of multiple node arrivals, the finger table updates would be costly. Therefore, Chord prefers to update finger tables lazily. Similar to the `stabilize()` function, each node n runs the `fix_fingers()` function periodically. It picks a finger randomly from the finger table at index i $(1 < i \leq l)$ and looks it up to find the true current successor of $n + 2^{i-1}$.

Node Failures

Chord addresses node failures on several levels. To detect node failures, all communication with other nodes needs to be checked for timeouts. When a node detects a failure of a finger during a lookup, it chooses the next best preceding node from its finger table. Since a short timeout is sufficient, lookup performance is not significantly affected in such a case. The `fix_fingers()` function ensures that failed nodes are removed from the finger tables. To expedite this process, `fix_fingers()` can be invoked specifically on a failed finger.

It is particularly important to maintain the accuracy of the successor information as the correctness of lookups depends on it. If, for example, the first three nodes in the finger table of node n fail simultaneously, the next live finger f might not be the true live successor s. Thus, node n would assume that a certain key k is located at f although it is located at s and would accordingly send incorrect replies to queries for k. The stabilization protocol can fail in a similar fashion when multiple nodes fail, even if live fingers are used as backups for failed successors.

To maintain a valid successor pointer in the presence of multiple simultaneous node failures, each node holds a successor list of length r. Instead of just a single successor pointer, it contains a node's first r successors. When a node detects the failure of its successor, it reverts to the next live node in its successor list. During `stabilize()`, a successor list with failed nodes is repaired by augmenting it with additional successors from a live node in the list. The Chord ring is affected only if all nodes from a successor list fail simultaneously.

The failure of a node not only means that it becomes unreachable but also that the data it managed is no longer available. Data loss from the failure of individual nodes can be prevented by replicating the data to other nodes. In Chord, the successor of a failed node becomes responsible for the keys and data of the failed node. Thus, an application utilizing Chord ideally replicates data to successor nodes. Chord can use the successor list to communicate this information and possible changes to the application.

Node Departures

Treating nodes that voluntarily leave a Chord network like failed ones does not affect the stability of the network. Yet it is inefficient because the failure needs to be detected and rectified. Therefore, a leaving node should transfer its keys to its successor and notify its successor and predecessor. This ensures that data is not lost and that the routing information remains intact.

8.2 Pastry

The Pastry distributed routing system was proposed in 2001 by Rowstron and Druschel [527]. Similar to Chord, its main goal is to create a completely decentralized, structured Peer-to-Peer system in which objects can be efficiently located and messages efficiently routed. Instead of organizing the identifier space as a Chord-like ring, the routing is based on numeric closeness of identifiers. In their work, Rowstron and Druschel focus not only on the number of routing hops, but also on network locality as factors in routing efficiency.

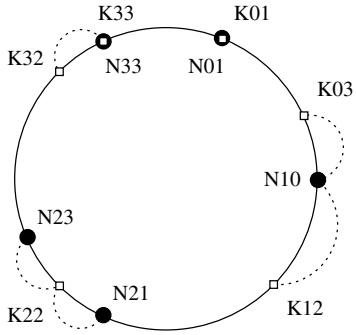


Fig. 8.2: A 4-bit Pastry identifier space with six keys mapped onto five nodes. Numeric closeness is an ambiguous metric for assigning keys to nodes as illustrated for key $K22$.

8.2.1 Identifier Space

In Pastry, nodes and data items uniquely associate with l-bit identifiers, i.e., integers in the range of 0 to $2^l - 1$ (l is typically 128). Under such associations, an identifier is termed a *node ID* or a *key*, respectively. Pastry views identifiers as strings of digits to the base 2^b where b is typically chosen to be 4. A key is located on the node to whose node ID it is numerically closest.

Figure 8.2 illustrates a Pastry identifier space with 4-bit identifiers and $b = 2$, so all numbers are to the base of 4. The closest node to, e.g., key $K01$ is $N01$, whereas $K03$ is located on node $N10$. The distances of key $K22$ to node $N21$ and $N23$ are equal so both nodes host this key to satisfy the requirements.

8.2.2 Routing Information

Pastry's node state is divided into three main elements. The *routing table*, similar to Chord's finger table, stores links into the identifier space. The *leaf set* contains nodes which are close in the identifier space (like Chord's successor list). Nodes that are close together in terms of network locality are listed in the *neighborhood set*.

Pastry measures network locality based on a given scalar network proximity metric. This metric is assumed to be already available from the network infrastructure and might range from IP hops to actual the geographical location of nodes.

Node 103220

Routing Table

0	031120	*1*	201303	312201
1	*0*	110003	120132	132012
2	100221	101203	102303	*3*
3		103112	*2*	103302
4		103210	*2*	
5	*0*			

Leaf Set

103123	103210	103302	103330

Neighborhood Set

031120	312201	120132	101203

Fig. 8.3: Pastry node state for the node 103220 in a 12-bit identifier space and a base of 4 ($l = 12$, $b = 2$). The routing table lists nodes with the length of the common node identifier prefix corresponding to the row index.

Routing Table

A Pastry node's routing table R (see Figure 8.3) is made up of $\frac{l}{b}$ rows with $2^b - 1$ entries per row (an additional column in Figure 8.3 also lists the digits of the local node ID for clarity). On node n, the entries in row i hold the identities of Pastry nodes whose node IDs share an i-digit prefix with n but differ in digit n itself. For example, the first row of the routing table is populated with nodes that have no prefix in common with n. When there is no node with an appropriate prefix, the corresponding table entry is left empty.

Routing tables built according to the Pastry scheme achieve an effect similar to Chord finger tables. A node has a coarse-grained knowledge of other nodes which are distant in the identifier space. The detail of the routing information increases with the proximity of other nodes in the identifier space. Without a large number of nearby nodes, the last rows of the routing table are only sparsely populated. Intuitively, the identifier space would need to be fully exhausted with node IDs for complete routing tables on all nodes. In a system with N nodes, only $log_{2^b}(N)$ routing table rows are populated on average.

In populating the routing table, there is a choice from the set of nodes with the appropriate identifier prefix. During the routing process, network locality can be exploited by selecting nodes which are close in terms of a network proximity metric.

Leaf Set

The routing table sorts node IDs by prefix. To increase lookup efficiency, the leaf set L of node n holds the $|L|$ nodes numerically closest to n. The routing

table and the leaf set are the two sources of information relevant for routing. The leaf set also plays a role similar to Chord's successor lists in recovering from failures of adjacent nodes.

Neighborhood Set

Instead of numeric closeness, the neighborhood set M is concerned with nodes that are close to the current node with regard to the network proximity metric. Thus, it is not involved in routing itself but in maintaining network locality in the routing information.

8.2.3 Routing Procedure

Routing in Pastry is divided into two main steps. First, a node checks whether the key k is within the range of its leaf set. If this is the case, it implies that k is located on one of the nearby nodes of the leaf set. Thus, the node forwards the query to the leaf set node numerically closest to k. In case this is the node itself, the routing process is finished.

If k does not fall into the range of leaf set nodes, the query needs to be forwarded over a longer distance using the routing table. In this case, a node n tries to pass the query on to a node which shares a longer common prefix with k than n itself. If there is no such entry in the routing table, the query is forwarded to a node which shares a prefix with k of the same length as n but which is numerically closer to k than n.

For example, a node with a routing table as in Figure 8.3 would send a query for key 103200 on to node 103210 as it is the leaf set node closest to the key. Since the leaf set holds the closest nodes, the key is known to be located on that node. A query for key 102022, although numerically closer to node 101203, is forwarded to node 102303 since it shares the prefix 102 with the key (in contrast to 10 as the current node does). For key 103000, there is no routing table entry with a longer common prefix than the current node. Thus the current node routes the query to node 103112 which has the same common prefix 103 but is numerically closer than the current node.

This scheme ensures that routing loops do not occur because the query is routed strictly to a node with a longer common identifier prefix than the current node, or to a numerically closer node with the same prefix.

8.2.4 Self-Organization

In practice, Pastry needs to deal with node arrivals, departures, and failures, while, at the same time, maintaining good routing performance if possible. This section describes how Pastry achieves these goals.

Node Arrival

Before joining a Pastry system, a node chooses a node ID. Pastry itself allows arbitrary node IDs, but applications may have more restrictive requirements. Commonly, a node ID is formed as the hash value of a node's public key or IP address.

For bootstrapping, the new node n is assumed to know a nearby Pastry node k based on the network proximity metric. Now n needs to initialize its node, i.e., its routing table, leaf and neighborhood set. Since k is assumed to be close to n, the nodes in k's neighborhood set are reasonably good choices for n, too. Thus, n copies the neighborhood set from k.

To build its routing table and leaf set, n needs to retrieve information about the Pastry nodes which are close to n in the identifier space. To do this, n routes a special "join" message via k to a key equal to n. According to the standard routing rules, the query is forwarded to node c with the numerically closest node ID. Due to this property, the leaf set of c is suitable for n, so it retrieves c's leaf set for itself.

The join request triggers all nodes, which forwarded the request towards c, to provide n with their routing information. Node n's routing table is constructed from the routing information of these nodes starting at row zero. As this row is independent of the local node ID, n can use the entries at row zero of k's routing table. In particular, it is assumed that n and k are close in terms of the network proximity metric. Since k stores nearby nodes in its routing table, these entries are also close to n. In the general case of n and k not sharing a common prefix, n cannot re-use entries from any other row in k's routing table.

The route of the join message from n to c leads via nodes $v_1 ... v_n$ with increasingly longer common prefixes of n and v_i. Thus, row 1 from the routing table of node v_1 is also a good choice for the same row of the routing table of n. The same is true for row 2 on node v_2 and so on. Based on this information, the routing table can be constructed for node n.

Finally, the new node sends its node state to all nodes in its routing data. These nodes can update their own routing information accordingly. In contrast to the lazy updates in Chord, this mechanism actively updates the state in all affected nodes when a new node joins the system. At this stage, the new node is fully present and reachable in the Pastry network.

The arrival and departure of nodes affects only a relatively small number of nodes in a Pastry system. Consequently, the state updates from multiple such operations rarely overlap and there is little contention. Thus, Pastry uses the following optimistic time-stamp-based approach to avoid major inconsistencies of node state: the state a new node receives is time-stamped. After the new node initializes its own internal state, it announces its state back to the other nodes including the original time-stamps. If the time-stamps do not match on the other nodes, they request the new node to repeat the join procedure.

Node Failure

Node failure is detected when a communication attempt with another node fails. Routing requires contacting nodes from the routing table and leaf set, resulting in lazy detection of failures. Since the neighborhood set is not involved in routing, Pastry nodes periodically test the liveness of the nodes in their neighborhood sets.

During routing, the failure of a single node in the routing table does not significantly delay the routing process. The local node can choose to forward the pending query to a different node from the same row in the routing table. Alternatively, a node could store backup nodes with each entry in the routing table.

Failed nodes need to be evicted from the routing table to preserve routing performance and correctness. To replace a failed node at entry i in row j of its routing table (R_j^i), a node contacts another node referenced in row i. Entries in the same row j of the the remote node are valid for the local node. Hence it can copy entry R_j^i from the remote node to its own routing table after verifying the liveness of the entry. In case it failed as well, the local node can probe the other nodes in row j for entry R_j^i. If no live node with the appropriate node ID prefix can be obtained in this way, the local node expands its horizon by querying nodes from the preceding row R_{j-1}. With very high probability, this procedure eventually finds a valid replacement for the failed routing table entry R_j^i, if one exists.

Repairing a failed entry in the leaf set L of a node is straightforward by utilizing the leaf sets of other nodes referenced in the local leaf set. The node contacts the leaf set entry with the largest index on the side of the failed node in order to retrieve the remote leaf set L'. If this node is unavailable, the local node can revert to leaf set entries with smaller indices. Since the entries in L' and L are close to each other in the identifier space and overlap, the node selects an appropriate replacement node from L' and adds it to its own leaf set. In the event that the replacement entry failed as well, the node again requests the leaf sets of other nodes from its local leaf set. For this procedure to be unsuccessful, $\frac{|L|}{2}$ adjacent nodes need to fail simultaneously. The probability of such a circumstance can be kept low even with modest values of $|L|$.

Nodes recover from node failures in their neighborhood sets in a fashion similar to repairing the leaf set. However, failures cannot be detected lazily since the nodes in the neighborhood set are not contacted regularly for routing purposes. Therefore, each node periodically checks the liveness of nodes in its neighborhood set. When a node failure is detected, a node consults the neighborhood sets of other neighbor nodes to determine an appropriate replacement entry.

Node Departure

Since Pastry can maintain stable routing information in the presence of node failures, deliberate node departures were originally treated as node failures for simplicity. However, a Pastry network would benefit from departure optimizations similar to those proposed for Chord. The primary goals would be to prevent data loss and reduce the amount of network overhead induced by Pastry's failure recovery mechanisms.

Arbitrary Failures

The approaches proposed for dealing with failures assumed that nodes fail by becoming unreachable. However, failures can lead to a random behavior of nodes, including malicious violations of the Pastry protocol. Rowstron and Druschel propose to amend these problems by statistically choosing alternative routes to circumvent failed nodes. Thus, a node chooses randomly, according to the constraints for routing correctness, from a set of nodes to route queries to with a bias towards the default route. A failed node would thus be able to interfere with some traffic but eventually be avoided after a number of retransmissions. How node arrivals and departures can be made more resilient to failed or malicious nodes is not addressed in the original work on Pastry.

8.2.5 Routing Performance

Pastry optimizes two aspects of routing and locating the node responsible for a given key: it attempts both to achieve a small number of hops to reach the destination node, and to exploit network locality to reduce the overhead of each individual hop.

Route Length

The routing scheme in Pastry essentially divides the identifier space into domains of size 2^n where n is a multiple of 2^b. Routes lead from high-order domains to low-order domains, thus reducing the remaining identifier space to be searched in each step. Intuitively, this results in an average number of routing steps related to the logarithm of the size of the system. This intuition is supported by a more detailed analysis.

It is assumed that routing information on all nodes is correct and that there are no node failures. There are three cases in the Pastry routing scheme, the first of which is to forward a query according to the routing table. In this case, the query is forwarded to a node with a longer prefix match than the current node. Thus, the number of nodes with longer prefix matches is

reduced by at least a factor of 2^b in each step, so the destination is reached in $log_{2^b}(N)$ steps.

The second case is to route a query via the leaf set. This increases the number of hops by one.

In the third case, the key is neither covered by the leaf set nor does the routing table contain an entry with a longer matching prefix than the current node. Consequently, the query is forwarded to a node with the same prefix length, adding an additional routing hop. For a moderate leaf set of size $|L| = 2 * 2^b$, the probability of this case occurring is less than 0.6% so it is very unlikely that more than one additional hop is incurred.

As a result, the complexity of routing remains at $O(log_{2^b}(N))$ on average. Higher values of b lead to faster routing but also increase the amount of state that needs to be managed at each node. Thus, b is typically 4 but Pastry implementations can choose an appropriate trade-off for the specific application.

Locality

By exploiting network locality, Pastry routing optimizes not only the number of hops but also the costs of each individual hop. The criteria to populate a node's routing table allow a choice among a number of nodes with matching ID prefixes for each routing table entry. By selecting nearby nodes in terms of network locality, the individual routing lengths are minimized. This approach does not necessarily yield the shortest end-to-end route but leads to reasonable total route lengths.

Initially, a Pastry node uses the routing table entries from nodes on a path to itself in the identifier space. The proximity of the new node n and the existing well-known node k implies that the entries in k's first row of the routing table are also close to n. The entries of subsequent rows from nodes on the path from k to n may seem close to k but not necessarily to n. However, the distance from k to these nodes is relatively long compared to the distance between k and n. This is because the entries in later routing table rows have to be chosen from a logarithmically smaller set of nodes in the system. Hence, their distance to k and n increases logarithmically on average. Another implication of this fact is that messages are routed over increasing distances the closer they get to the destination ID.

8.3 Content Addressable Network *CAN*

Ratnasamy et al. presented their work on scalable content-addressable networks [505] in 2001, the same year in which Chord and Pastry were introduced. In CAN, keys and values are mapped onto numerically close nodes. Locating objects and routing messages in CAN is very simple as it requires

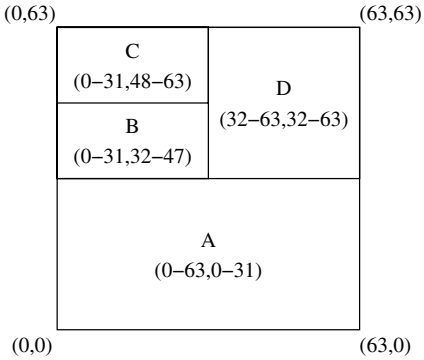


Fig. 8.4: A two-dimensional six-bit CAN identifier space with four nodes. For simplicity, it is depicted as a plane instead of a torus.

knowledge only about a node's immediate neighbors. However, CAN introduces the notion of multi-dimensional identifier spaces by which routing efficiency is greatly improved compared to linear neighbor traversal in a single dimension. CAN generalizes the Chord and Pastry approaches in certain areas and introduces design optimizations also applicable to other DHT systems.

8.3.1 Identifier Space

A CAN identifier space can be thought of as a d-dimensional version of a Chord or Pastry identifier space. Each data item is assigned an identifier, e.g., of the form $< x, y, z >$ for $d = 3$. All arithmetic on identifiers is again performed modulo the largest coordinate in each dimension. The geometrical representation of a CAN identifier space is thus a d-torus. The original work on CAN suggests a space with continuous coordinates between 0.0 and 1.0 but it also applies to discrete coordinate spaces.

The identifier space in CAN is partitioned among the participating nodes as shown in Figure 8.4. Each node is said to *own* a *zone*, i.e., a certain part of the identifier space. CAN ensures that the entire space is divided into non-overlapping zones. In Figure 8.4, a two-dimensional identifier space (represented by a plane instead of a torus for simplicity) is divided into four zones. In contrast to Chord and Pastry, CAN does not assign a particular identifier to a node. Instead, the extent of its zone is used to locate and identify a node.

As typical for $(key, value)$ pairs in DHTs, CAN keys are derived from the value (or a representation of it) by applying a uniform hash function. The easiest way of deriving multi-dimensional identifiers from flat hash values is to apportion a fixed set of bits to the coordinate in each dimension. For example, a 160-bit hash value would be divided into two 80-bit segments

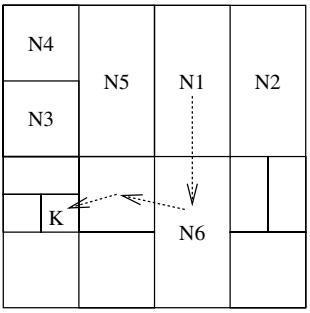


Fig. 8.5: The route from node $N1$ to a key K with coordinates (x, y) in a two-dimensional CAN topology before node $N7$ joins. Neighbor set of $N1$: $\{N2, N6, N5\}$

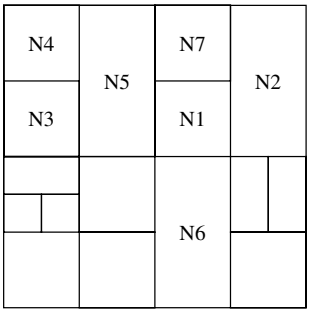


Fig. 8.6: New node $N7$ arrives in the zone of $N1$. $N1$ splits its zone and assigns one half to $N7$. Updated neighbor set of $N1$: $\{N7, N2, N6, N5\}$.

which represent $< x, y >$ coordinates in a two-dimensional identifier space. Because a key represents a point P in the identifier space, $(key, value)$ pairs are stored on the node owning the zone which covers P.

8.3.2 Routing Information

For routing purposes, a CAN node stores information only about its immediate neighbors. Two nodes in a d-dimensional space are considered neighbors if their coordinates overlap in one dimension and are adjacent to each other in $d-1$ dimension. Figure 8.5 illustrates neighbor relationships. For example, node $N1$ and $N6$ are neighbors as they overlap in the y dimension and are next to each other in the x dimension. At the same time, node $N5$ and $N6$ are not neighbors as they do not overlap in any dimension. Similarly, node $N1$ and $N4$ overlap in the x dimension but are not adjacent in the y dimension, so they are not neighbors of each other.

The routing information in CAN is comprised of the IP address, a port, and the zone of every neighbor of a node. This data is necessary to access the CAN service on a neighbor node and to know its location in the identifier space. In a d-dimensional identifier space partitioned into zones of equal size, each node has $2d$ neighbors. Thus, the number of nodes participating in a CAN system can grow very large while the necessary routing information per node remains constant.

8.3.3 Routing Procedure

Conceptually, routing in CAN follows a straight line in the cartesian identifier space from the source to the destination coordinates. Each node contributes to this process utilizing its neighbor information.

Each CAN message contains the destination coordinates. If the local node does not own the zone that includes these coordinates, it forwards the message to the neighbor with the coordinates closest to the destination, as illustrated in Figure 8.5. In a d-dimensional space equally partitioned into n zones, this procedure results in an average of $O((d/4)(n^{\frac{1}{d}}))$ routing steps. This expresses the intuitive consequence that increasing the number of dimensions significantly reduces the average route length.

8.3.4 Self-Organization

CAN dynamically organizes nodes into an overlay network which implements the operations in the identifier space. It assigns zones of the identifier space to individual nodes in such a way that zones do not overlap and there are no gaps in the identifier space. This partitioning needs to be robust when nodes join or leave a CAN system or when they fail.

Node Arrival

A node n joining a CAN system needs to be allocated a zone and the zone neighbors need to learn of the existence of n. The three main steps in this procedure are: to find an existing node of a CAN system; to determine which zone to assign to the new node; and to update the neighbor state.

Like Chord and Pastry, CAN is not tied to a particular mechanism for locating nodes in the overlay network to be joined. However, Ratnasamy et al. suggest using a dynamic DNS name to record one or more nodes belonging to a particular CAN system. The referenced nodes may in turn publish a list of other nodes in the same CAN overlay. This scheme allows for replication and randomized node selection to circumvent node failures.

Given a randomly chosen location in the identifier space, the new node n sends a special *join* message via one of the existing nodes to these coordinates. Join messages are forwarded according to the standard CAN routing procedure. After the join message reaches the destination node d, d splits its zone in half and assigns one half to n (cf. Figure 8.6). In order to ease the merging of zones when nodes leave and to equally partition the identifier space, CAN assumes a certain ordering of the dimensions by which zones are split. For example, zones may be split along the first (x) dimension, then along the second (y) dimension and so on. Finally, d transfers the $(key, value)$ pairs to n for which it has become responsible.

Node n and d exchange neighborhood information such that n learns of its neighbors from d and d adds n to its own set of neighbors. Then node n immediately informs all its neighbors of its presence. Through *update* messages, every node in the system also provides its direct neighbors periodically with its own neighborhood and zone information. Thus, only a small region of the identifier space is affected by node arrival. Its size depends on the number of dimensions but stays constant with the total number of nodes in the system.

Node Failure

The zones of failing or leaving nodes must be taken over by live nodes to maintain a valid partitioning of the CAN identifier space. A CAN node detects the failure of a neighbor when it ceases to send update messages. In such an event, the node starts a timer. When the timer fires, it sends *takeover* messages to the neighbors of the failed node. The timer is set up such that nodes with large zones have long timeouts while small zones result in short timeouts. Consequently, nodes with small zone sizes send their takeover messages first.

When a node receives a takeover message, it cancels its own timer provided its zone is larger than the one advertised in the message. Otherwise, it replies with its own takeover message. This scheme efficiently chooses the neighboring node with the smallest zone volume. The elected node claims ownership of the deserted zone and merges it with its own zone if possible. Alternatively, it temporarily manages both zones.

The hash-table data of a failed node is lost. However, the application utilizing CAN is expected to periodically refresh data items it inserted into the DHT (the same is true for the other systems presented here). Thus, the hash table state is eventually restored.

During routing, a node may find that the neighbor to which a message is to be forwarded has failed and the repair mechanism has not yet set in. In such a case, it forwards the message to the live neighbor next closest to the destination coordinates. If all neighbors failed, which are closer to the destination, the local node floods the message in a controlled manner within the overlay until a closer node is found.

Node Departure

When a node l deliberately leaves a CAN system, it notifies a neighbor n whose zone can be merged with l's zone. If no such neighbor exists, l chooses the neighbor with the smallest zone volume. It then copies the contents of its hash table to the selected node so this data remains available.

As described above, departing and failing nodes can leave a neighbor node managing more than one zone at a time. CAN uses a background process

which reassigns zones to nodes to prevent fragmentation of the identifier space.

8.3.5 Routing Performance

CAN comes with a number of design optimizations which focus both on reducing the number of hops in a CAN overlay and on lowering path latencies. In combination, these steps result in a significant overall improvement of routing performance.

Increasing the number of dimensions in the identifier space reduces the number of routing hops and slightly increases the amount of neighbor state to store on each node. The average path length in a system with n nodes and d dimensions scales as $O(d(n^{\frac{1}{d}}))$. Higher dimensionality also improves routing fault tolerance because each node has a larger set of neighbors to choose from as alternatives to a failed node.

CAN also supports multiple instances of a DHT in different coordinate spaces termed *realities*. A node is present in all identifier spaces and owns a different zone in each of them. In each reality, all DHT data is replicated and distributed among the nodes. Thus, a system with r realities implies that each node manages r different zones and neighbor sets, one for each reality.

With multiple realities, data availability is improved through the replication of data in each reality. Also, a node has more options to route around failed neighbors. Furthermore, a node can choose the shortest route from itself to a destination in all realities. Thus, the average length of routes is reduced significantly. The different advantages and per-node state requirements of multiple dimensions and realities need to be traded off against each other based on application requirements.

The nonconformance of a CAN overlay and the underlying IP infrastructure may lead to significantly longer route lengths than direct IP routing. Hence, CAN suggests incorporating routing metrics which are not based on the distance between two nodes in the identifier space. For example, each node could measure the round-trip time (RTT) to neighboring nodes and use this information to forward messages to those neighbors with the best ratio of RTT and ID space distance to the destination. Experiments show an improvement of 24% to 40% in per-hop latency with this approach [505].

Another optimization is to *overload* a zone by allowing multiple nodes to manage the same zone. This effectively reduces the number of nodes in the system and thus results in fewer routing hops. Each node also forwards messages to the neighboring node with the lowest RTT in a neighbor zone, thus reducing per-hop latency. Finally, when the DHT data for a zone is replicated among all nodes of that zone, all these nodes need to fail simultaneously for the data to be lost.

With multiple hash functions, a $(key, value)$ pair would be associated with a different identifier per hash function. Storing and accessing the $(key, value)$ at each of the corresponding nodes increases data availability. Furthermore, routing can be performed in parallel towards all the different locations of a data item reducing the average query latency. However, these improvements come at the cost of additional per-node state and routing traffic.

The mechanisms presented above reduce per-hop latency by increasing the number of neighbors known to a node. This allows a node to forward messages to neighbors with a low RTT. However, CAN may also construct the overlay so it resembles more closely the underlying IP network. To place nodes close to each other, both at the IP and the overlay level, CAN assumes the existence of well-known landmark nodes. Before joining a CAN network, a node samples its RTT to the landmarks and chooses a zone close to a landmark with a low RTT. Thus, the network latency en route to its neighbors can be expected to be low resulting in lower per-hop latency.

For a more uniform partition of the identifier space, nodes should not join a CAN system at a random location. Instead, the node which manages the initial random location queries its neighbors for their zone volume. The node with the largest zone volume is then chosen to split its zone and assign half of it to the new node. This mechanism contributes significantly to a uniform partitioning of the coordinate space.

For real load-balancing, however, the zone size is not the only factor to consider. Particularly popular $(key, value)$ pairs create hot spots in the identifier space and can place substantial load on the nodes hosting them. In a CAN network, overload caused by hot-spots may be reduced through caching and replication. Each node caches a number of recently accessed data items and satisfies queries for these data items from its cache if possible. Overloaded nodes may also actively replicate popular keys to their neighbors. The neighbors in turn reply to a certain fraction of these frequent requests themselves. Thus, load is distributed over a wider area of the identifier space.

8.4 Symphony

The Symphony protocol can be seen as a variation of Chord that exploits the small world phenomenon. As described by Manku et al. in [400], it is of constant degree because each node establishes only a constant number of links to other nodes. In contrast, Chord, Pastry, and CAN require a number of links which depends on the total number of nodes in the system. This basic property of Symphony significantly reduces the amount of per-node state and network traffic when the overlay topology changes. However, with an increasing number of nodes, it does not scale as well as Chord.

Like Chord, the identifier space in Symphony is constructed as a ring structure and each node maintains a pointer to its successor and predeces-

sor on the ring. In Symphony, the Chord finger table is replaced by a constant but configurable number k of long distance links. In contrast to other systems, there is no deterministic construction rule for long distance links. Instead, these links are are chosen randomly according to harmonic distributions (hence the name *Symphony*). Effectively, the harmonic distribution of long-distance links favors large distances in the identifier space for a system with few nodes and decreasingly smaller distances as the system grows.

The basic routing in this setup is trivial: a query is forwarded to the node with the shortest distance to the destination key. By exploiting the bi-directional nature of links to other nodes, routing both clockwise and counter-clockwise leads, on average, to a 25% to 30% reduction of routing hops. Symphony additionally employs a *1-lookahead* approach. The lookahead table of each node records those nodes which are reachable through the successor, predecessor, and long distance links, i.e., the neighbors of a node's neighbors. Instead of routing greedily, a node forwards messages to its direct neighbor (not a neighbor's neighbor) which promises the best progression towards the destination. This reduces the average number of routing hops by 40% at the expense of management overhead when nodes join or leave the system.

In comparison with the systems discussed previously, the main contribution of Symphony is its constant degree topology resulting in very low costs of per-node state and of node arrivals and departures. It also utilizes bi-directional links between nodes and bi-directional routing. Symphony's routing performance ($O(\frac{1}{k}log^2(N))$) is competitive compared with Chord and the other systems ($O(log(N))$) but does not scale as well with exceedingly large numbers of nodes. However, nodes can vary the number of links they maintain to the rest of the system during run-time based on their capabilities, which is not permitted by the original designs of Chord, Pastry, and CAN.

8.5 Viceroy

In 2002, Malkhi et al. proposed Viceroy [399], another variation on Chord. It improves on the original Chord algorithm through a hierarchical structure of the ID space with constant degree which approximates a butterfly topology. This results in less per-node state and less management traffic but slightly lower routing performance than Chord.

Like Symphony, Viceroy borrows from Chord's fundamental ring topology with successor and predecessor links on each node. It also introduces a new node state called a *level*. When joining the system, a node chooses a random level in the range from 1 to $log(N)$. Thus, the Viceroy topology can be thought of as $log(N)$ vertically stacked rings. However, the node ID still serves as the unique identifier for nodes so that no two nodes may occupy the same node ID, regardless of their level.

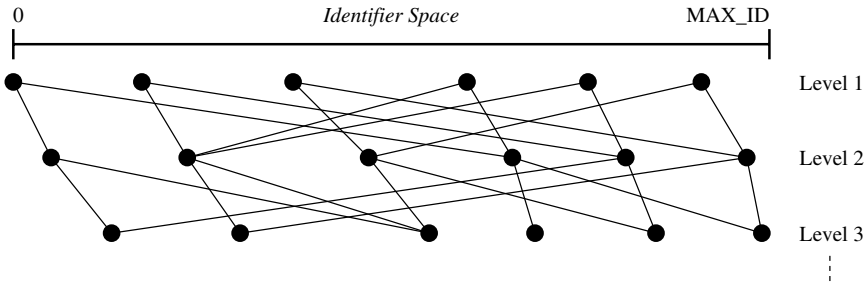


Fig. 8.7: A Viceroy topology with 18 nodes. Lines indicate short- and long-range downlinks; other links and lower levels are omitted for simplicity.

A Viceroy node n maintains a total of seven links to other nodes, independent of the network size. As n's two closest neighbors in the ID space, i.e., its successor and predecessor, might reside on any level, it also establishes a *level-ring link* to each of the closest nodes clockwise and counter-clockwise on its own level l. In order to connect to other levels, n creates an uplink to a nearby node at level $l-1$ and a short-range and long-range downlink to level $l+1$. The long-range downlink is chosen such that it connects to a node at a distance of roughly $\frac{1}{2^l}$. Thus, the distance covered by the long-range links is reduced logarithmically with lower levels as depicted in Figure 8.7.

The routing procedure is split into three phases closely related to the available routing information. First, a query is forwarded to level one along the uplinks. Second, a query recursively traverses the downlinks towards the destination. On each level, it chooses the downlink which leads to a node closer to the destination, without overshooting it in the clockwise direction. After reaching a node without downlinks, the query is forwarded along ring-level and successor links until it reaches the target identifier. The authors of Viceroy show that this routing algorithm yields an average number of $O(log(N))$ routing hops.

Like Symphony, Viceroy features a constant degree linkage in its node state. However, every node establishes seven links whereas Symphony keeps this number configurable even at run-time. Furthermore and similar to Chord, the rigid layout of the identifier space requires more link updates than Symphony when nodes join or leave the system. At the same time, the scalability of its routing latency of $O(log(N))$ surpasses that of Symphony, while not approaching that of Chord, Pastry, and CAN.

8.6 Kademlia

In their work on Kademlia [405], Maymounkov and Mazières observe a mismatch in the design of Pastry: its routing metric (identifier prefix length) does

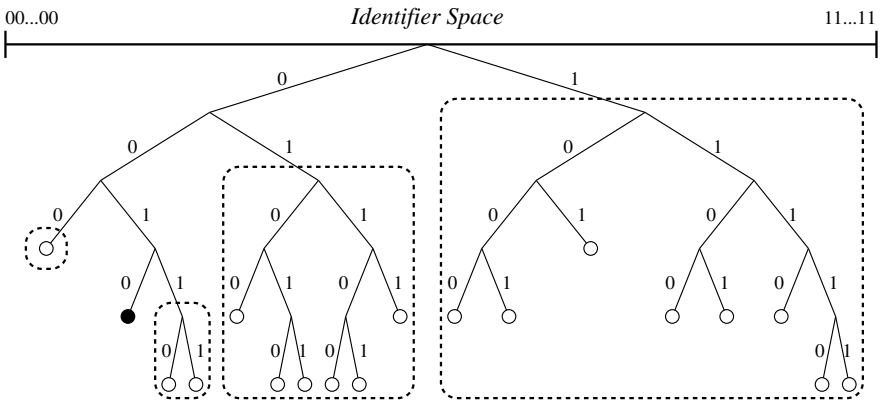


Fig. 8.8: An example of a Kademlia topology. The black node 0010 knows about the subtrees that do not match its identifier as indicated by the dotted squares. Each node successively forwards a query to α nodes in a destination subtree.

not necessarily correspond to the actual numeric closeness of identifiers. As a result, Pastry requires two routing phases which impacts routing performance and complicates formal analysis. Thus, Kademlia uses an XOR routing metric which improves on these problems and optionally offers additional parallelism for lookup operations.

Kademlia's XOR metric measures the distance between two identifiers i and j by interpreting the result of the bit-wise exclusive OR function on i and j as an integer. For example, the distance between the identifiers 3 and 5 is 6. Considering the shortest unique prefix of a node identifier, this metric effectively treats nodes and their identifiers as the leaves of a binary tree. For each node, Kademlia further divides the tree into subtrees not containing the node, as illustrated in Figure 8.8.

Each node knows of at least one node in each of the subtrees. A query for an identifier is forwarded to the subtree with the longest matching prefix until the destination node is reached. Similar to Chord, this halves the remaining identifier space to search in each step and implies a routing latency of $O(log(N))$ routing hops on average.

In many cases, a node knows of more than a single node per subtree. Similar to Pastry, the Kademlia protocols suggests forwarding queries to α nodes per subtree in parallel. By biasing the choice of nodes towards short round-trip times, the latency of the individual hops can be reduced. With this scheme, a failed node does not delay the lookup operation. However, bandwidth usage is increased compared to linear lookups.

When choosing remote nodes in other subtrees, Kademlia favors old links over nodes that only recently joined the network. This design choice is based on the observation that nodes with long uptime have a higher probability

of remaining available than fresh nodes. This increases the stability of the routing topology and also prevents good links from being flushed from the routing tables by distributed denial-of-service attacks, as can be the case in other DHT systems.

With its XOR metric, Kademlia's routing has been formally proved consistent and achieves a lookup latency of $O(log(N))$. The required amount of node state grows with the size of a Kademlia network. However, it is configurable and together with the adjustable parallelism factor allows for a trade-off of node state, bandwidth consumption, and lookup latency.

8.7 Summary

The core feature of every DHT system is its self-organizing distributed operation. All presented systems aim to remain fully functional and usable at scales of thousands or even millions of participating nodes. This obviously implies that node failures must be both tolerated and of low impact to the operation and performance of the overall system. Hence, performance considerations are an integral part of the design of each system.

Since the lookup of a key is probably the most frequently executed operation and essential to all DHT systems, a strong focus is put on its performance. The number of routing hops is an important factor for end-to-end latency, but the latency of each hop also plays an important role. Generally, additional routing information on each node also provides a chance for choosing better routes. However, the management of this information and of links to other nodes in a system also incurs overhead in processing time and bandwidth consumption.

System	Routing Hops	Node State	Arrival	Departure
Chord	$O(\frac{1}{2}log_2(N))$	$O(2log_2(N))$	$O(log_2^2(N))$	$O(log_2^2(N))$
Pastry	$O(\frac{1}{b}log_2(N))$	$O(\frac{1}{b}(2^b-1)log_2(N))$	$O(log_{2^b}(N))$	$O(log_b(N))$
CAN	$O(\frac{D}{2}N^{\frac{1}{D}})$	$O(2D)$	$O(\frac{D}{2}N^{\frac{1}{D}})$	$O(2D)$
Symphony	$O(\frac{c}{k}log^2(N))$	$O(2k+2)$	$O(log^2(N))$	$O(log^2(N))$
Viceroy	$O(\frac{c}{k}log^2(N))$	$O(2k+2)$	$O(log_2(N))$	$O(log_2(N))$
Kademlia	$O(log_b(N))$	$O(b \cdot log_b(N))$	$O(log_b(N))$	$O(log_b(N))$

Table 8.1: Performance comparison of DHT systems. The columns show the averages for the number of routing hops during a key lookup, the amount of per-node state, and the number of messages when nodes join or leave the system.

Table 8.1 summarizes the routing latency, per-node state, and the costs of node arrivals and departures in the systems discussed above. It illustrates how design choices, like a constant-degree topology, affect the properties of a system. It should be noted that these results are valid only for the original proposals of each system and that the $O()$ notation leaves ample room for variation. In many cases, design optimizations from one system can also be transferred to another system. Furthermore, the effect of implementation optimizations should not be underestimated. The particular behavior of a DHT network in a certain application scenario needs to be determined individually through simulation or real-world experiments.