

# Chapter 2

## DHT Theory

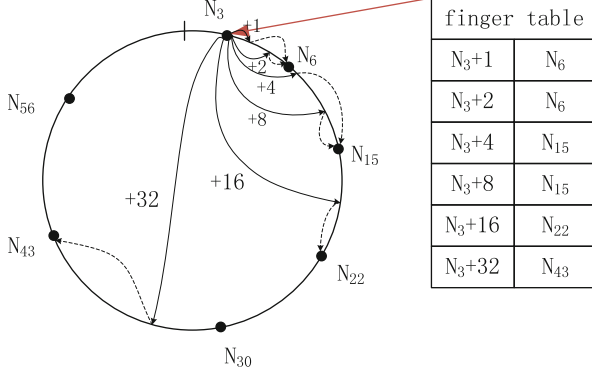
In this chapter, we discuss a set of DHT variants, which widely influence the design and development of distributed systems in recent years. In each section, we first describe the structure of each variant, present the key elements such as the routing model and the data model, and then discuss the solutions to a common problem in distributed environments [20], namely, nodes dynamically join and leave distributed systems. In the last section, we compare the DHT variants from numerous aspects, such as overlay network topology, distance metric, routing and data model and so on.

### 2.1 Chord

Chord [6] is a distributed lookup protocol. It solves a general but fundamental problem in P2P networks, namely, how to efficiently locate the node which stores a particular data item. In addition, Chord is designed to address challenges in P2P systems and applications, for instance, load balancing, decentralization, scalability, availability and flexible naming.

In Chord, the DHT space is a circle, which is a one-dimensional space referred to as the *Chord ring*. Although the structure of Chord is a ring-like consistent hashing, their aims are different. Consistent hashing focuses on the caching problem, while Chord is an architecture for organizing the nodes and contents in P2P networks. In Chord, both nodes and data are mapped into this space by a pre-determined hash function (e.g., SHA-1 [21]). The keys used to map nodes and data are referred to as the identifiers (IDs) of the corresponding nodes and data. IDs for nodes can be generated by applying a hash function to unique information of individual nodes (e.g., nodes' IP addresses), and IDs for data can be computed by applying a hash function to the data themselves.

IDs are ordered on the Chord ring by calculating  $ID \bmod 2^m$ , where  $m$  is the number of bits in the key. Therefore, IDs correspond to points on the Chord ring. All IDs are arranged clockwise in an ascending order on the Chord ring. For a node  $N_i$  with its ID being  $i$ , we define its previous node on the clockwise ring as



**Fig. 2.1** An example of finger table

predecessor( $N_i$ ), and define its next node as successor( $N_i$ ). In particular, the node with the maximal ID (i.e.,  $N_{ID_{\max}}$ ) chooses the node with the minimal ID (i.e.,  $N_{ID_{\min}}$ ) as its successor. Additionally,  $N_{ID_{\max}}$  is the predecessor of  $N_{ID_{\min}}$ .

However, if each node only knows its predecessor and successor in the one-dimensional Chord ring (which is a directed graph), such a DHT system would be inefficient and vulnerable for numerous reasons. First, the time complexity of DHT lookup is  $O(n)$ , where  $n$  is the number of peers in such a system; hence, the complexity could be too high when  $n$  is large. Second, each node can only send messages to its successor on the ring; hence, the node connectivity is 1 in such a system (both in-degree and out-degree). Third, since each node has only one choice for routing, if one node fails, the connectivity of the graph will be destroyed.

Chord introduces a *finger table* structure to solve the above problems. Each node has a finger table, and each finger table maintains up to  $m$  nodes (recall that the size of the key space is  $2^m$ ) for the purpose of efficient routing and increasing the connectivity of the graph. For an arbitrary node  $N$ , the  $i$ th entry in its finger table contains the first node clockwise from  $N + 2^{i-1}$ :

$$N.\text{finger}[i] = \text{successor}(N + 2^{i-1}). \quad (2.1)$$

Each node maintains the data whose IDs are in the range between this node's ID and its predecessor's ID. The finger table structure improves the connectivity of the Chord ring and thus improves the routing on the ring, which in turn significantly reduces the complexity. More specifically, the time complexity of the DHT lookup operation is reduced from  $O(n)$  to  $O(\log n)$  due to increasing the connectivity from 1 to  $O(m)$ .

Figure 2.1 shows an example of the Chord ring where  $m = 6$  and the finger table of node  $N_3$ . The first entry (i.e., the zeroth entry) in this finger table contains  $N_6$ , since according to Eq. 2.1, the ID of the first neighboring node defined by the finger table should be  $(3 + 2^0) \bmod 2^6 = 4$ . However, there does not exist a node whose ID is 4. Therefore, the immediate next node on the ring, i.e.,  $N_6$ , is chosen for the

reason that  $\text{successor}(4) = N_6$ . Similarly, the last entry (i.e., the fifth entry) in the finger table contains  $N_{43}$ , because  $(3 + 2^5) \bmod 2^6 = 35$  and the first node that succeeds the node  $N_{35}$  is  $\text{successor}(35) = 43$ .

The two primitives for data storage and retrieval are as follows: (1)  $\text{put}(k, v)$  stores a given data  $v$  whose ID is  $k$  on the node that has an ID closest to  $k$ , and (2)  $v = \text{get}(k)$  retrieves the corresponding data  $v$  stored at a node using the ID  $k$ . The common key to both data storage and retrieval is how to locate the node that is responsible for storing the data with a given ID. Chord locates the node as follows. When a node  $N$  with ID  $j$  needs to locate where the data with ID  $k$  is stored (or should be stored), it would send a query to the node  $N'$  satisfying the Eq. 2.2:

$$N' = \begin{cases} N.\text{finger}[0] & d(k, j) \leq d(N.\text{finger}[0], j) \\ N.\text{finger}[i] & d(k, j) \leq d(N.\text{finger}[i+1], j) \text{ and } d(k, j) > d(N.\text{finger}[i], j) \\ N.\text{finger}[m-1] & \text{otherwise} \end{cases} \quad (2.2)$$

In Eq. 2.2,  $N.\text{finger}[0]$  is the ID of node  $N$ , and the distance between ID  $x$  and ID  $y$  is  $d(x, y) = (x - y) \bmod 2^m$ , which is the Euclidean distance in the one-dimensional ring space. Note that during data storage and retrieval, each node always tries to send queries to the node that is the closest to the querying node.

A node may dynamically join or leave a Chord system. The primitive  $\text{join}()$  inserts a new joining node into the Chord ring and updates relevant nodes' successors accordingly. When a node joins a Chord system, it is assumed to know at least one node on the Chord ring, which helps the new node to locate its successor. Moreover, a stabilization protocol runs periodically to update the successor lists and finger tables. The primitive  $\text{leave}()$  removes a voluntarily leaving node from the Chord ring and updates the lists of successors and finger tables accordingly. When a node leaves or fails, some other node may lose its successor (if the leaving/failing node is the successor). To mitigate this situation, each node maintains a list of the first  $r$  successors. When one of its successors leaves or fails, a node simply chooses the next node on this list as the successor. By tuning the parameter of  $r$ , Chord could balance the robustness and the cost of maintaining the successor list.

Researchers have studied how to improve Chord extensively and there has been a large body of literature on Chord; To name a few of such studies, Flocchini et al. [22] proposed a method that combines multiple Chord rings to achieve data redundancy and reduce the average routing path length; Joung et al. [23] proposed a two-layer structure called *Chord*<sup>2</sup>, where super peers are introduced to construct a conduct ring which could reduce the maintenance cost; Kaashoek et al. introduced a Chord-like structure Koorde [24] where the bruijn graphs [25] substitute the finger table; Cordasco et al. [26] proposed a family of Chord-based P2P schemes, F-Chord( $\alpha$ ), using the Fibonacci numbers to improve the degree, diameter and average path length of Chord. Furthermore, H-F-Chord( $\alpha$ ) using the NoN (Neighbors of Neighbors) technique [27–29] is more efficient in terms of its average path length  $O(\log n / \log \log n)$ ; Ganesan et al. [30] optimizes Chord routing algorithms by exploiting the bidirectional edges, both clockwise and counterclockwise, which reduces the average routing path length.

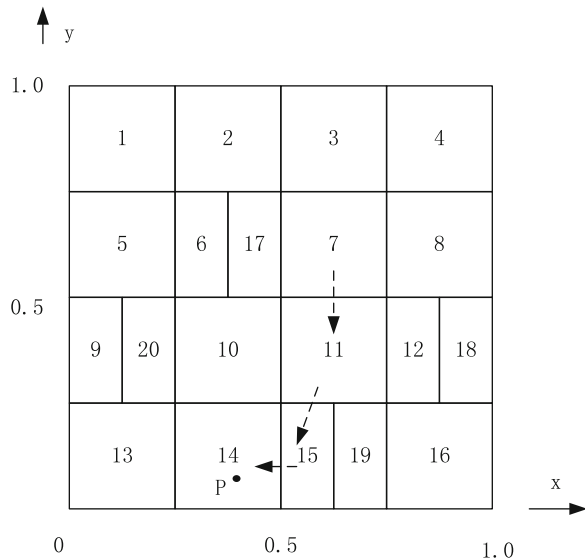
## 2.2 Content-Addressable Network (CAN)

CAN [31] is a distributed, Internet-scale, DHT-based infrastructure that provides hash table-like functionalities. Different from the one-dimensional space in Chord, the DHT space in CAN is a  $d$ -dimensional Cartesian space. The  $d$ -dimensional space is further dynamically partitioned among all nodes and each node only maintains its own individual and distinct zone in the space.

In CAN, every node maintains  $2d$  neighbors, thus the node connectivity is  $2d$ . Here the notion of “neighbors” means two zones that overlap along  $d - 1$  dimensions and have neighbors on one dimension. As a result, CAN does not need to introduce complex structures such as long links (e.g., the finger table in Chord) connecting nodes, which are further away from each other in the  $d$ -dimensional space, in order to improve the connectivity and reduce the complexity of routing. Note that  $d$  is a constant independent from the number of nodes in the system, which means that the number of neighbors each node maintains is a constant, no matter how many nodes the CAN system may have.

Figure 2.2 illustrates a two-dimensional CAN with 20 nodes. Each dimension covers  $[0, 1)$  and every node maintains a zone in the grid. For example, node 1 maintains the zone  $(0 - 0.25, 0.75 - 1.0)$ , and node 17 maintains the zone  $(0.375 - 0.5, 0.5 - 0.75)$ . Every node maintains the IDs (e.g., IP addresses) of its neighbors that maintain the zones in the neighborhood.

The routing in CAN works as follows. When receiving a message with a specific destination, a node routes the message towards the destination using a simple greedy algorithm, i.e., the node goes through the list of its neighbors to select the one that is closest to the destination, and then forwards the message to the selected neighbor.



**Fig. 2.2** Example of a two-dimensional CAN with 20 nodes

This greedy forwarding process continues until the message arrives to the designated destination. For instance, Fig. 2.2 shows a routing path from the node 7 to the point  $P$  in the zone maintained by the node 14. The dashed lines illustrate the steps in which nodes greedily forward a message from the source (i.e., node 7) to the destination  $P$ .

The data in CAN is stored and retrieved based on the notion of key-value pairs, similar to the notion adopted in Chord. More specifically, in order to store a key-value pair  $(k, v)$  where  $k$  is the key and  $v$  is the corresponding value (i.e., data),  $k$  is first mapped to a point  $P$  in the  $d$ -dimensional space using a hash function specifically chosen by CAN; then the corresponding  $(k, v)$  pair is delivered to and stored on the node that maintains the zone where  $P$  is located. Similarly, in order to retrieve the value  $v$  for a given key  $k$ , a node should first obtain the point  $P$  by mapping the key  $k$ , and then retrieve the corresponding value  $v$  from the node that maintains the zone where  $P$  is located.

The protocol that accommodates the dynamic node arrival and departure is more complex than that in Chord, due to the fact that the  $d$ -dimensional space adopted by CAN is more complex than the one-dimensional space by Chord. When a node  $i$  joins the CAN system, it should be introduced into the system by another node  $j$  that is already in the system. More specifically, the new joining node must send a JOIN request to find an existing node whose zone  $z_j$  can be split. This zone should be split by a certain number of dimensions, in such a way that the zone can be reclaimed when nodes leave in the future. When a node  $i$  leaves the system, the zone  $z_i$  that it maintains should be taken over by other remaining nodes. If the zone  $z_i$  can be merged with a zone  $z_j$  which is maintained by a node  $j$  (one of the neighbors of the leaving node  $i$ ), then the node  $j$  should reclaim the zone  $z_i$  by merging it with its own zone  $z_j$  and maintain the new larger zone. Otherwise, the zone  $z_i$  should be taken over by the neighbor whose zone is the smallest.

For instance, suppose that the first splittable dimension is the  $x$  axis. When a new node (ID is 21) is joining the system, it finds that the zone maintained by the node 2 can be split in half. After the split, the zone maintained by the node 2 changes from  $(0.25 - 0.5, 0.75 - 1.0)$  to  $(0.25 - 0.375, 0.75 - 1.0)$ . The other half, i.e.,  $(0.375 - 0.5, 0.75 - 1.0)$ , will be assigned to the joining node 21. When the node 17 leaves, its zone should be reclaimed and merged with the zone maintained by the node 6. Note that after the zones split (and merge), the neighbors of the affected zones should update their neighbor lists so that the joining node (and departing node) can participate in (and be removed from) the routing system. For example, before the node 17 joins in the CAN system, the neighbor list of the node 6 is  $\{2, 5, 7, 10\}$  and the list of the node 7 is  $\{3, 6, 8, 11\}$ . After the node 17 successfully joins, the zone maintained by the node 6 is split into two smaller zones, one is maintained by the node 6, the other is by the node 17. Then node 6, 17 and the neighbors 2, 5, 7, 10 should update their neighbor lists. After updating the neighbor lists, the neighbors of the node 6 are  $\{2, 5, 10, 17\}$ , and the neighbors of the node 7 are  $\{3, 8, 11, 17\}$ .

To maintain the healthiness of the CAN system, each node periodically sends update messages to its neighbors. When a node has not received the update messages

from one of its neighbors for a long time (longer than a certain timeout threshold), it considers that the neighbor fails and starts a takeover timer for it. When the timer expires, the node sends a TAKEOVER message to all neighbors of the failed nodes, announcing that it takes over the zone that is formerly maintained by the failed node. However, if the node receives a TAKEOVER message before the timer expires, the node then cancels this timer. When such a timer is initialized, it should be set up in a way that it is proportional to the failed node's zone. More specifically, if one node fails, the neighbor who maintains the smallest zone should send the TAKEOVER message at the earliest moment and therefore takes over the failing node's zone.

Besides, to improve the robustness and performance of the CAN system, three technologies, i.e. increasing dimensions, multiple realities and RTT-weighted routing, are introduced [31]. As a result, CAN behaves well in large-scale distributed systems and has been applied to many applications in such systems.

### 2.3 Global Information Sharing Protocol (GISP)

GISP [32] is a protocol for fully decentralized P2P networks. GISP does not make any assumption on the network structure, thus it is applicable to both structured and unstructured P2P networks.

There are a set of key principles that guide the design of GISP. First, each node should maintain as much peer information as possible, so the network has great connectivity. Second, each node should discard information of unreachable peers and keep more information about nodes that are numerically closer. Last but not least, each node may possess different levels of capability; however, the more capability one node possesses, the greater responsibility the node should take. GISP introduces the notion of "peer strength" to quantify the capability of a node. The powerful nodes (with higher peer strength) should keep more data and have more connectivity than those with less peer strength.

The routing model in GISP works as follows. GISP leverages hash functions such as MD5 and SHA-1 to map any binary data including a keyword into a number with fixed bit length. Similar to Chord and CAN, each node in GISP has a unique ID in the hash space. GISP defines the distance between two nodes by  $\text{distance}(i, j) / (2^{s_i-1} 2^{s_j-1})$ , where  $i$  and  $j$  are the IDs of two nodes,  $s_i$  and  $s_j$  are the values of the two nodes' "peer strength." GISP adopts a greedy routing strategy, namely, when forwarding a message to its destination, a node selects the next-hop node who has the shortest distance to the destination.

The data storage and retrieval are similar to Chord. When inserting the data  $v$  into a GISP system, the key  $k$  is computed (i.e., the hash value of the data  $v$ ) and the node whose ID is numerically closest to  $k$  is selected to maintain the data. In other words, GISP selects the node who has the shortest distance to the hash value of the data. For example, suppose that a GISP system has four nodes, whose ID are 100, 110, 115, 122 respectively. If a piece of data with key 107 is pushed in the system, then the node with ID is 110 would be selected to maintain the data.

Data retrieval is similar to the data storage process. More specifically, given a key  $k$ , the node that is responsible for maintaining the data is located and queried, then the data  $v$  will be routed back to the requesting node. Data storage and retrieval are conceptually straightforward; however, it is difficult to completely delete any stored data. In GISP, when a node stores a piece of data, it also set up a timer for the data; each node periodically check the data it maintains and delete the expired data.

Since node failures in P2P networks are not uncommon, in GISP the data is duplicated and the replicas are distributed to multiple nodes whose IDs are closer to the hash value of the corresponding data. The number of data replicas is determined either statically or dynamically. The larger the number is, the more robust the system is to multiple node failures, and the more storage capacity is required. As a result, routing messages are not sent to only one node but to a group of peers (in the ascending order of the distance from the nodes to the hash value of the data). In order to avoid routing loops, each message is associated with a list of peers to which this message has already been sent. For example, suppose that the node 4 is sending a message  $M$  to the nodes 5, 6 and 7. When node 4 sends  $M$  to node 7, it also tells node 7 that it has already sent this message  $M$  to node 5 and 6. Thus, node 7 will not route  $M$  to 5 and 6.

When a node  $i$  is joining a GISP system, it is assumed that node  $i$  knows at least an active node  $j$  in the system. From the node  $j$ , node  $i$  acquires knowledge about other nodes in the network. When a node  $i$  leaves the system, it notifies other peers its departure. In the case that some nodes leave the system without notifying others (e.g., due to the network connectivity problems), GISP can still work well unless too many peers fail at the same time and as a result too much data stored in the network is lost.

GISP introduces a latency-based mechanism to relate the overlay network topology with the real underlying physical network topology. Such a mechanism can reduce the cost of network routing in GISP. More specifically, when a new node is joining the network, GISP first determines this node's ID based on the latency values of the existing nodes that it knows. By doing so, nodes that are closer in the underlay network topology are likely to form clusters (i.e., the distances/latencies between nodes in such clusters are lower) in the overlay network topology.

## 2.4 Kademlia

Kademlia [33] is a P2P storage and lookup system. Both nodes and data in Kademlia are assigned with 160-bit integer IDs. More specifically, each node chooses a random 160-bit integer as its ID, and data is stored in the form of key-value pairs, where the key is a 160-bit value generated by hash functions such as SHA-1 and being the ID of the value, and the value is the data stored in Kademlia.

Unlike Chord and CAN, Kademlia defines the distance between two nodes  $i$  and  $j$  by the bitwise exclusive OR operation (XOR), i.e.,  $d(i, j) = i \oplus j$ . This distance metric is unidirectional like the metric used in Chord, which means for any given key  $i$  and a distance  $l > 0$ , there are only one key  $j$  that satisfies  $d(i, j) = l$ .

Every piece of data in the form of key-value pair is stored on  $k$  nodes whose IDs are the closest to the key. Here  $k$  is a key parameter in Kademlia to determine data redundancy and system stability.

Each node  $i$  in Kademlia maintains multiple  $k$ -buckets. Each  $k$ -bucket is a linked list with a maximum length of  $k$ . Each  $k$ -bucket keeps a list of nodes, which are sorted in the ascending order of recent activities, i.e., the node that is the least recently seen is stored at the head, and the node that is the most recently seen is stored at the tail. The node whose distance from the node  $i$  is in the range of  $[2^m, 2^{m+1}]$  is stored in the  $m$ th  $k$ -bucket (note that  $0 \leq m < 160$ ). The nodes in the  $k$ -buckets are regarded as the neighbors of the node  $i$ .

Unlike in Chord and CAN, a node updates its neighbors dynamically upon receiving any messages from them. More specifically, when a node  $i$  receives a message from another node  $j$ , which is located in the  $m$ th  $k$ -bucket, this  $k$ -bucket of node  $i$  will be updated in the following way. If  $j$  already exists in the  $k$ -bucket,  $i$  moves  $j$  to the tail of the list, as node  $j$  is the node that is the most recently seen. If  $j$  is not in the  $k$ -bucket and the bucket has fewer than  $k$  nodes, node  $i$  just inserts  $j$  at the tail of the list. If the bucket is full,  $i$  pings the node at the head of this  $k$ -bucket. If this head node responds, node  $i$  moves it to the tail and ignores node  $j$ . Otherwise,  $i$  removes the head node and inserts  $j$  at the tail.

Kademlia has four RPC-like primitives, i.e., PING, STORE, FIND\_NODE and FIND\_VALUE. The PING primitive probes a node to check whether it is online or not. The STORE primitive is used to store a key-value pair. The FIND\_NODE primitive finds a set of nodes that are closest to a given node; in other words, it returns  $k$  nodes from one or multiple  $k$ -buckets, whose IDs are closest to the given node's 160-bit ID. The FIND\_VALUE primitive behaves like FIND\_NODE, except that it returns the stored value. These primitives work in a recursive way, and in order to improve the efficiency of Kademlia, a lookup procedure is invoked by the FIND\_NODE and FIND\_VALUE primitives. More specifically, at the beginning, the lookup initiator picks  $\alpha$  nodes from its closest  $k$ -bucket and sends multiple parallel FIND\_NODE requests to these  $\alpha$  nodes. If the proper node is not found, the initiator re-sends the FIND\_NODE to the nodes it just learned in the last recursive execution. A key-value pair may be stored on multiple nodes. With the recursive lookup procedure, the key-value pair spreads across the network every hour. This method ensures that for any data, multiple replicas exist for robustness. Every key-value pair is deleted 24 h after it is initially pushed into the network.

When a node  $i$  is joining the network, it is assumed that it knows a node  $j$  which is active and already in Kademlia. The joining process consists of multiple steps. First, node  $i$  inserts  $j$  into its  $k$ -buckets. Second, node  $i$  starts a node lookup procedure for its own ID, from which  $i$  learns some of the new nodes. Finally, node  $i$  updates the  $k$ -buckets. During this process, node  $i$  strengthens its  $k$ -buckets and inserts itself into other nodes'  $k$ -buckets. When a node fails or leaves, it does not notify any other node. There is no need for a special procedure to cope with node departures, as the mechanism of  $k$ -buckets ensures that the leaving nodes will be removed from the  $k$ -buckets.



For the reasons of the simple distance metric and the  $k$ -bucket mechanism, Kademlia becomes the most widely used DHT system—it has been adopted by many popular P2P applications such as Overnet [34], eDonkey™/eMule™ [35] and BitTorrent™ [36, 37]. Researchers have made many efforts on analyzing and improving the lookup performance of the Kademlia protocol [38–42], in order to enhance the practicality of the Kademlia-based system.

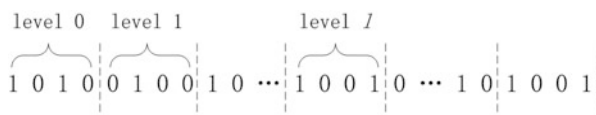
## 2.5 Pastry

Pastry [43] is a self-organizing overlay network with a targeted basic capability of routing messages efficiently. Every node in Pastry has a 128-bit ID. A node ID is divided into multiple levels, each of which represents a domain. Each domain is represented by  $b$  (an integer by which 128 is divisible) contiguous bits in the node ID, i.e., the domain at level  $l$  is specified by the bits at positions  $b \times l$  to  $b \times (l+1) - 1$ . Each level contains  $2^b$  domains numbered from 0 to  $2^b - 1$ . Figure 2.3 illustrates an example of dividing the Pastry node ID with  $b = 4$ . The first four bits specify the domain at level 0, and the following four bits specify the domain at level 1. In this case the domain at level  $l$  is domain 9 (i.e., the binary bit string is 1001).

### Routing Model

Each node has a routing table, a neighborhood set and a namespace set. The routing table of a node contains  $2^b - 1$  nodes for each level  $l$ ; these nodes have the same prefix up to level  $l - 1$  as the local node. Hence, the routing table contains  $L \times 2^b - 1$  nodes, where  $L$  is the number of the levels. The neighborhood set contains  $M$  nodes, which are closest to the local node (measured by their physical distances). However, note that the neighborhood set is not used in routing messages. The namespace set contains  $L$  nodes which are closest to and centered around the local node. The namespace set is used during the message routing and object insertion.

When a node routes an incoming message, the node first checks if the destination's ID falls in its namespace set. If so, the message will be sent directly to the destination node. Otherwise, the node uses the routing table to choose the domain at a level  $l$ , where the nodes at this level  $l$  share the longest prefix with the destination node's ID. Then the node selects a node in this domain as the next hop. The selected node has to be alive and be closer to the destination than other nodes in the same domain.



**Fig. 2.3** Node ID division in Pastry with  $b = 4$

## Data Model

Every data object  $v$  in Pastry has an object ID  $k$  that is at least 128 bits long, which can be generated by a hash function. When storing an object  $v$  into the Pastry network, Pastry routes a message containing the data object  $v$  to the node whose ID is numerically closest to  $k$  (i.e., the ID of  $v$ ). In order to improve the data availability, each data object is not only stored on one node but also on a set of extra nodes whose IDs are numerically close to the object ID.

When a node  $i$  joins a Pastry system, it is assumed that the node knows at least an active node  $j$  in the network. Node  $j$  routes a “join” message on behalf of node  $i$  and the message is destined for node  $i$ . Eventually, the message will be routed to a node  $i'$  whose ID is numerically closest to  $i$ . Then node  $i$  copies node  $j$ ’s neighborhood set as its initial neighborhood set, and takes the namespace set of node  $i'$  as its initial namespace set. Node  $i$  also initializes its routing table using the relevant information of the nodes on the path from  $j$  to  $i'$ , including  $j$  and  $i'$ . When a node  $i$  is leaving a Pastry system, there is no specific protocols to handle the node departure. Rather, nodes solve this problem by refreshing their routing table, neighborhood set and namespace set.

## 2.6 Tapestry

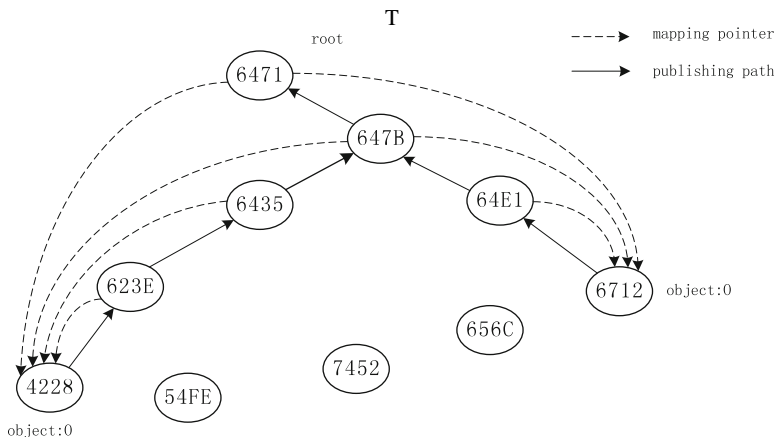
Tapestry [44] is a P2P overlay routing infrastructure which offers scalable, location-independent and efficient routing of messages using only local resources.

In Tapestry, each node has a 160-bit ID, and each application-specific endpoints (e.g., objects) is assigned with a 160-bit globally unique identifier (GUID), both of which can be generated by using a hash function such as SHA-1. The “distance” between two nodes is assessed digit by digit; for example, a node whose ID is “2341” which is closer to the node “2347” than the node “2338.” Below, we will elaborate the routing and data model of Tapestry, respectively.

### Routing Model

Each node maintains a routing table which consists of a set of neighboring nodes. Tapestry guarantees that any node can be reached from the root in at most  $\log_{\beta} N$  hops, where  $N$  is the size of the name space and  $\beta$  is the base of IDs. In order to do so, neighboring nodes in the routing table are organized into  $\log_{\beta} N$  levels. In the  $j$ th level, at most  $c \times \beta$  pointers to the relative nodes that begins with  $\text{prefix}(N, j - 1)$  are maintained, where  $c$  neighbors differ only on the  $j$ th digit. For instance, “325A” is maintained in the fourth level of the routing table of “3259” with  $\beta = 16$ .

When a node routes an incoming message, the node selects the next-hop node from the neighbors by matching the level prefix, which is similar to the longest



**Fig. 2.4** An example of storing (publishing) a data object in Tapestry

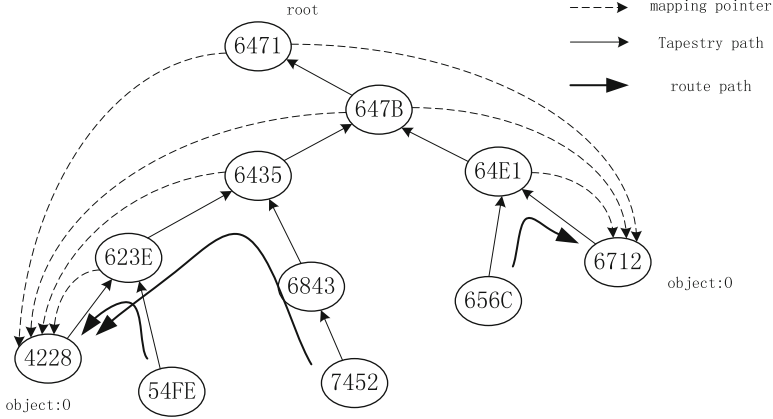
prefix routing method used in the CIDR IP address allocation architecture [45]. As a result, the IDs of the nodes on a route vary gradually (e.g.,  $3*** \Rightarrow 34** \Rightarrow 34A* \Rightarrow 34AE$ , where “\*” is the wildcard). Tapestry also provides the surrogate routing to route messages to some active nodes with similar IDs. This scheme can mitigate the problem of single-point failures.

## Data Model

Each object in Tapestry has a root, as Tapestry maps each data object to a root node whose ID is equal or closest to the GUID of the object. When a node stores or retrieves a data object, it always sends the request to the root of the object. More specifically, when a node  $i$  publishes a data object  $v$  it stores, it sends the publishing message to the root of the object  $O$ . Each of the intermediate nodes on the path from  $i$  to the root node stores a location mapping  $(v, i)$ . Upon receiving a request message for the object  $v$ , each node on the path to the root node checks if it has the location mapping for  $v$ . If it does have the mapping, it redirects the message to the node  $i$  who stores  $v$ ; otherwise, it forwards the message to the next hop towards the root node. As a result, for any data object, the routing paths of the request messages form a unique spanning tree, with the root being the root of the object.

Figure 2.4 illustrates an example of publishing data objects in Tapestry. Both the node “4228” and the node “6712” store a data object  $O$ . Note that the node “6471” is the root for this data object. When the node “4228” and “6712” send publishing messages, each of the intermediate nodes along the paths towards to the object’s root creates a location mapping, i.e., a pointer for the object  $O$  to the publishers “4228” and “6712.”

Figure 2.5 illustrates an example of how nodes retrieve a data object  $O$ . There are three nodes, “54FE,” “7452” and “656C,” retrieving the data object  $O$  by sending querying messages towards the root of this object. When the message from “54FE”



**Fig. 2.5** An example of retrieving a data object in Tapestry

is received by “623E,” the latter finds that it has maintained the location mapping for the object  $O$  (which points to “4228”), and thus forwards the querying message directly to the publisher “4228,” rather than further forwards the message to the root “6471.” Similarly, the messages from “7452” and “656C” are sent directly to “4228” and “6712” respectively. By doing so, Tapestry efficiently improves the routing throughput and latency.

When a node  $i$  is joining a Tapestry system, it is assumed that it knows (or can find) an active node which is already in the system. The data objects which should be rooted at node  $i$  must be migrated to  $i$ . During the migration, node  $i$  constructs its own routing table, and notifies other nodes so that they can insert  $i$  into their routing table.

When a node  $i$  is leaving the system, Tapestry solves the problem in two cases: voluntary node deletion and non-voluntary node deletion. In the first case, the leaving node  $i$  notifies all nodes that are related to  $i$  and moves the objects it maintains to the new root. In the second case, node  $i$  leaves or fails without any notification. In this case, nodes rely on periodical messages to detect whether the outgoing link and node fails. Furthermore, Tapestry builds redundant pointers in routing tables to improve robustness to node departures. A combination of these techniques retains nearly 100 % success rate for routing messages.

## 2.7 Viceroy

Viceroy [46] is constructed on the butterfly network. In Viceroy each node has two associated values:  $id$  and  $level$ .  $id$  is the identity of the node in the network, which is a random value in the range  $[0, 1)$  and is fixed throughout its participation;  $level$  is a positive integer in the range  $[1, \log N]$  and changes as the network evolves.

Nodes in Viceroy form three types of sub-topologies: a *general ring*, which is similar to the ring in Chord and is constructed by the successor and predecessor relationships, multiple *level rings*, where all nodes in the same level form a ring, and a *butterfly* network where each node connects to proper nodes to form a butterfly network. In the butterfly network, each non-leaf node at level  $l$  connects to two nodes at the lower level  $l + 1$ ; such links are referred to as the left and the right down links respectively. The left down link connects to the node which is the clockwise closest, and the right down link connects to the node at about  $1/2^l$  away at level  $l + 1$ . Additionally, each node at level  $l > 1$  also connects to one node at the upper level  $l - 1$  which is numerically the closest node to the local node (such links are referred to as up links).

The routing model and the data model are unified via a key primitive, LOOKUP, in Viceroy. This primitive is used not only to maintain the network during nodes' joining and leaving, but also to data retrievals. The primitive LOOKUP consists of three phases. Firstly, the root node at level 1 is found through the up link. Secondly, the next-hop node is selected using the down links. When a node at level  $l$  calculates the distance between the destination and itself to determine which of the two down links (i.e., left or right) should be used to forward messages. More specifically, if it finds that the distance is at least  $1/2^l$ , then it forwards the lookup message through the right down link; otherwise, the left down link should be used. Thirdly, when a message reaches a node without down links or overshooting the destination, the message is forwarded along the level ring until the destination node is found.

When a node  $i$  is joining the Viceroy system, it should join each of the three types of sub-networks. First, node  $i$  finds out its predecessor and successor in the general ring based on its ID. It inserts itself into this ring and gets the key-value pairs between  $i$  and  $\text{predecessor}(i)$  from its successor. This procedure operates similar to Chord. Second, node  $i$  selects its level by a level selection mechanism and joins the level ring. Last, node  $i$  joins the butterfly by choosing the two down links and one up link.

When a node is leaving the Viceroy system, it removes all of the outbound connections and notifies all neighboring nodes so that they can find a replacement. Moreover, the leaving node must transfer the resources it maintains to its successor.

## 2.8 Comparison

All DHT variants discussed in the chapter focus on how to efficiently manage a large number of nodes and data in a distributed system, and each variation has its unique characteristics.

In this section, we will discuss their similarities and differences from the perspectives of how the overlay network is constructed, the distance metric, the routing and data model, and how node dynamics are handled.

### 2.8.1 *Overlay Network Topology*

Chord is the simplest variation of DHT. It is a one-dimensional ring where all IDs of both data and nodes are arranged clockwise in the ascending order. Every node maintains the data whose IDs fall in the range bounded by the node's ID and its predecessor's ID. CAN is featured with a  $d$ -dimensional hyper-cube structure. Its  $d$ -dimensional Cartesian space is divided into a number of non-overlapping zones, each of which is maintained by one node. Any data object is mapped into the  $d$ -dimensional hyper-cube as a point in one of the zones. GISP is a structureless variation of DHT, where there is no limitation for the number of connections among nodes. Any two nodes have a direct connection if they know each other and they are alive. Kademlia, Pastry and Tapestry all form a one-dimensional structure which could be thought of as a tree structure. The nodes' identifiers constitute the leaf nodes in the tree. They use fixed-length bit strings as the IDs for nodes and data objects (Kademlia and Tapestry use 160-bit IDs; Pastry uses 128-bit node ID and the object ID is at least 128 bits long). Additionally, Kademlia organizes the network with the XOR operation, so its structure is a special tree referred to as the XOR-tree [47]. Pastry uses a ring structure to assist the routing when the tree structure can't find a proper target. Viceroy constructs a butterfly structure, which is the most complex one in the aforementioned variations.

The levels of connectivity available in these variants also differ significantly. More specifically, CAN has strong connectivity between nodes due to the  $d$ -dimensional hyper-cube structure, and Viceroy benefits from the property of butterfly structure, while in all other variants, nodes are less connected. Thus nodes in these variants with less connectivity have to maintain multiple other nodes as their neighbors (these nodes are not its numerical neighbors). For instance, each node in Chord maintains a finger table, containing  $O(\log N)$  nodes which are not the numerical neighbors on the one-dimensional ring. The distances from this node to the nodes in the finger table are half of ring perimeter, one-quarter of ring perimeter, one-eighth of ring perimeter,  $\dots$ , so on and so forth. As a comparison, Viceroy maintains three types of connectivity (i.e., links on the general ring, a level ring and a butterfly network) which allow nodes to connect not only to their numerical neighbors but also to other nodes. Due to the butterfly structure, each node in Viceroy only maintains  $O(1)$  peers. The enhancement of the connectivity in these structures significantly improves the efficiency of routing.

### 2.8.2 *Distance*

The distance metric,  $d(i, j)$ , measures the distance between two nodes (or data objects) of  $i$  and  $j$  in a distributed system. This metric is the key and basis for routing and data retrieval operations. As a result, the difference in the distance metrics used by different variants leads to different routing strategies.

In Chord, the IDs of nodes and data objects are treated in the same manner. The distance from  $i$  to  $j$  is defined as  $\bar{d}(i, j) = (j - i) \bmod 2^m$ , where  $m$  is the number of bits in the key and node identifiers. In CAN, the distance is the Euclidean distance in the  $d$ -dimension space. From the geometric point of view, it is the distance between two points in the  $d$ -dimensional hyper-cube. In GISP, the distance between two data objects is the differences between their IDs. For the distance between two nodes, GISP introduces a new parameter called “peer strength,” which represents the capability of a node. More specifically, the distance between two nodes with IDs  $i$  and  $j$  is  $\bar{d}(i, j)/(2^{s_i-1}2^{s_j-1})$ , where  $s_i$  and  $s_j$  are the “peer strength” values of these two nodes. This distance metric assigns a greater responsibility to the nodes with a more powerful capability. In Kademlia, the distance between node  $i$  and  $j$  is the bitwise exclusive OR (XOR), i.e.,  $\bar{d}(i, j) = i \oplus j$ . This metric has extremely low computational complexity than the Euclidean distances (e.g., adopted by Chord and CAN), and it does not need an additional algorithmic structure for discovering the target in the nodes which share the same prefix. In Pastry and Tapestry, the distances both assessed digit by digit as in Plaxton [48]. In Viceroy, the IDs of data objects and nodes are numbers in the range  $[0, 1)$ , thus the distance from node  $i$  to  $j$  is  $\bar{d}(i, j) = (j - i) \bmod 1$ , similar to the distance metric used in Chord.

The distance metrics of Chord, Kademlia and Viceroy are unidirectional, which means that for any given key  $k$  and a distance  $D > 0$ , only one key  $k'$  can satisfy  $\bar{d}(k, k') = D$ . In other systems the number of nodes satisfying this condition is more than 1. This also means that Chord, Kademlia and Viceroy can determine a unique node by distance for the routing purpose, while all remaining variants need to determine which node is the next hop by additional parameters. Besides, The distance metrics used in Chord and Viceroy are directed, which means that  $\bar{d}(i, j) \neq \bar{d}(j, i)$  in most situations.

### 2.8.3 Routing and Data Model

We compare the routing and data models adopted by different variants from three perspectives: the routing table, the routing path selection and length. Below we assume that there are  $N$  nodes in the network.

#### Routing Table

Each node in all DHT variants maintains the information of a set of other nodes for the purpose of routing. However, different variants maintain different types of nodes.

In Chord, each node stores three lists of nodes: a predecessor node, a finger table consisting of up to  $m$  entries/nodes (note that the size of the ID space is  $2^m$ ), and a successor node list with  $r$  entries. The total number of nodes maintained by each node is  $O(\log N)$ . In CAN, each node in the  $d$ -dimensional hyper-cube maintains

a list of  $2d$  neighbors, which is independent of the total number of nodes in the system (i.e.,  $N$ ). This property is desirable since nodes keep a constant number of neighbors regardless of the system size. In Kademlia, the length of node IDs is 160 bits. For any  $i$  ( $0 \leq i < 160$ ), each node keeps a  $k$ -bucket, in which at most  $k$  nodes are stored. The distance from any of the nodes in the  $i$ th  $k$ -bucket to the local node is in the range  $[2^i, 2^{i+1}]$ . Therefore, each node in Kademlia maintains at most  $160 \times k$  neighboring nodes, which is equal to  $O(\log N)$  in essence. In GISP, each node maintains as many nodes as possible, but GISP does not explicitly define strategies for managing such information. In Pastry, each node maintains three sets of nodes: a routing table consisting of approximately  $\lceil \log_{2^b} N \rceil \times (2^b - 1)$  nodes (recall that the levels are denoted by  $b$  bits), a neighborhood set of  $M$  nodes, and a namespace set of  $L$  nodes. The typical values for  $L$  and  $M$  are  $2^b$  and  $2 \times 2^b$ , respectively. In Tapestry, each node has a location mapping which consists of a neighbor map with  $\log_{2^b} N$  levels. To some extent, this is similar to the routing table in Pastry. Each level contains  $c \times 2^b$  nodes, where  $c$  is chosen for redundancy of routing nodes. The number of nodes every node maintains in these two variations is  $O(\log_{2^b} N)$ . Furthermore, Tapestry nodes also maintain pointers to the nodes who publishes the objects to the local nodes, which reduces the lookup time greatly. In Viceroy, each node maintains a constant (small) number of nodes. The number of nodes a node maintains is at most 7.

In CAN and Viceroy, each node maintains only a constant number of nodes, regardless the size of the network. On the contrary, in most of other variants, each node maintains  $O(\log N)$  nodes, which is a varying number dependent on the size of the network. In summary, Viceroy nodes maintain the least number of nodes [i.e.,  $O(1)$ ], while others maintain  $O(\log_2 N)$ . Thus Viceroy costs least to maintain the complete structure; however, the structure of Viceroy is more sensitive to node failures than other variants.

## Path Length

The average length of routing paths is an important parameter that suggests the efficiency of routing: the less number of hops a routing path has, the more efficient the routing is.

In Chord, the length of the routing paths is no more than  $O(\log N)$  hops, as a result of the finger table which allows to search in an exponential manner. In CAN, due to the  $d$ -dimensional hyper-cube structure, the average routing path length is  $\frac{d}{4} N^{\frac{1}{d}}$  and any node can be reached by another node in  $O(dN^{\frac{1}{d}})$  hops. In Kademlia, Pastry and Tapestry, the average length of routing paths can be improved to  $O(\log_{2^b} N)$ , where  $b$  bits are merged to denote a digit. For example,  $b = 4$  means that the IDs are hexadecimal and the 160-bits ID is regarded as a 20-digit hexadecimal number. when  $b = 1$ , Kademlia, Pastry and Tapestry are the same as Chord in terms of the efficiency of routing. In Viceroy, the average length of routing paths is also  $O(\log N)$ . In GISP, since any two nodes may be connected, it is challenging to analyze its average routing path length of GISP.



## Path Selection

Each node in the above DHT variants knows only partial information about the network. However, DHT ensures efficient routing path selection with such partial information.

More specifically, each node tries its best to find out the node closest to the destination. Here the “closest” node is chosen based on nodes’ numerical distances to the destination. The distance metrics used in different variants are different; as a result, the paths from the same source node to the destination are different in different variants. All DHT variants adopt a locally optimal solution in each step, which may achieve the global optimal solution. This is the reason why DHT-based overlay networks perform efficiently in large-scale distributed systems.

### 2.8.4 Node’s Joining and Leaving

When a node  $i$  wants to join the system, it is assumed that it knows at least one active peer  $j$  which is already in the system. In Chord, the active node  $j$  helps node  $i$  to find its successor. Node  $i$  joining the system requires that some nodes in the system should update their finger tables. With a stabilization protocol running periodically, nodes update their successor nodes and their finger tables gradually, which costs  $O(\log N)$  times and  $O(\log_2 N)$  messages. In CAN the active node  $j$  introduces node  $i$  into the system which allocates a zone for  $i$  to maintain. Since all nodes only maintain a list of neighboring nodes, when a node joins in CAN, the neighbors of this new node should update their neighbor lists. This is sufficient to generate the correct routing path without a stabilization protocol. Thus the joining operation in CAN costs  $O(1)$  times. In Kademlia, the joining node  $i$  starts a LOOKUP operation to insert itself into other nodes’  $k$ -buckets and constructs its own  $k$ -buckets gradually. In GISP, the network needs extra time to stabilize when a node comes. In Pastry and Tapestry, node  $i$  must try its best to learn enough number of peers and tells others to memorize it. In Viceroy, node  $i$  constructs the three types of links by the JOIN operation [46].

When a node  $i$  leaves the network, affected nodes must update the node information they maintain. Chord runs a stabilization protocol to update the nodes’ successors and finger tables. In CAN, each normal node sends update messages to its neighbors periodically. When neighbors find a node has left, one of them takes over the failed node’s zone and notifies other neighbors. In GISP, when a node finds that some nodes are unreachable, it deletes these nodes from its routing list. In Kademlia and Pastry, there are no specific protocol to handle node departures; rather, they detect the target node before routing. This method reduces the impact of node leaving, but it may increase the latency of message routing. In Tapestry, nodes use heartbeat messages to detect whether a node is alive. In Viceroy, when a node leaves voluntarily, the leaving node should execute the LEAVE operation [46] before its departure. Otherwise, when node failures occur, the structure of Viceroy should be repaired by executing the LOOKUP operation (Table 2.1).

**Table 2.1** Comparison of classical DHT variants. (a) Comparison from topology, distance and path length (b) Comparison from path selection, number of peers, node's joining & leaving

(a)				
Algorithm	Overlay network topology	Distance from $i$ to $j$	Routing path length	
Chord	One-dimensional ring	$(j - i) \bmod 2^m$	$O(\log N)$	
CAN	$d$ -dimensional cube	Euclidean distance	$\frac{d}{4} N^{\frac{1}{d}}$	
GISP	Structureless	Objects: the difference of the two IDs; Nodes: $(i, j)/(2^{s_i}-1, 2^{s_j}-1)$ ; $s_i, s_j$ are “peer strength”	Uncertain	
Kademlia	XOR-tree	$i \oplus j$	$O(\log_{2^b} N)$	
Pastry	Tree+ring	Assessed digit by digit	$O(\log_{2^b} N)$	
Tapestry	Tree	Same as Pastry	$O(\log_{2^b} N)$	
Viceroy	Butterfly	$(j - i) \bmod 1$	$O(\log N)$	
(b)				
Algorithm	Routing path selection	# of maintained peers	Node’s joining	Node’s leaving
Chord	Greedy algorithm	$O(\log_2 N)$ construct	Find successor; protocol	Run stabilization
CAN		$2d$	Generate neighbor list	Update neighbor lists
GISP		As many as possible	Generate routing list	Delete failing nodes from routing list
Kademlia		At most $160 \times k(O(\log_2 N))$	Constructs $k$ -buckets	Detect the target node before routing
Pastry		$O(\log_{2^b} N)$	Generate routing table, neighborhood set and a namespace set	Detect the target node before routing
Tapestry		$O(\log_{2^b} N)$	Construct the routing table	Heartbreak message
Viceroy		At most 7	Construct the three kinds of links	Repaired by the LOOKUP operation

Distributed Hash Table

Theory, Platforms and Applications

Zhang, H.; Wen, Y.; Xie, H.; Yu, N.

2013, VIII, 67 p. 15 illus., 3 illus. in color., Softcover

ISBN: 978-1-4614-9007-4