7. Distributed Hash Tables

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

In the last few years, an increasing number of massively distributed systems with millions of participants has emerged within very short time frames. Applications, such as instant messaging, file-sharing, and content distribution have attracted countless numbers of users. For example, Skype gained more than 2.5 millions of users within twelve months, and more than 50% of Internet traffic is originated by BitTorrent. These very large and still rapidly growing systems attest to a new era for the design and deployment of distributed systems [52]. In particular, they reflect what the major challenges are today for designing and implementing distributed systems: *scalability*, *flexibility*, and *instant deployment*.

As already defined in Chapter 2, the Peer-to-Peer paradigm relies on the design and implementation of distributed systems where each system has (nearly) the same functionality and responsibility. By definition, these systems have to coordinate themselves in a distributed manner without centralized control and without the use of centralized services. Thus, scalability should be an inherent property of Peer-to-Peer systems. Unfortunately, not all of them have shown this to be true so far. In this chapter, we will demonstrate this by discussing the *lookup problem* – a fundamental challenge for all kinds of massively distributed and Peer-to-Peer systems.

First, we introduce the problem of managing and retrieving data in distributed systems, compare three basic approaches for this, and show that some of them do not scale well, even though they are Peer-to-Peer approaches. As a result, we introduce the promising concept of the Distributed Hash Table (DHT) for designing and deploying highly scalable distributed systems. In this chapter, we focus only on the basic properties and mechanisms of Distributed Hash Tables; specifics of certain DHT approaches are presented in the next chapter.

The remainder of this chapter is organized as follows. After discussing in Section 7.1 general concepts for distributed management and retrieval of data in Peer-to-Peer systems, the subsequent sections introduce Distributed Hash Tables, in particular their fundamentals (Section 7.2), the concept of content-based routing (Section 7.3), and DHT interfaces (Section 7.4). The next chapter presents specific algorithms of popular DHT approaches, e.g., how to organize the address space, and how routing in a Distributed Hash Table is performed. In Chapter 9, we discuss aspects of reliability and load-balancing in Distributed Hash Tables.

R. Steinmetz and K. Wehrle (Eds.): P2P Systems and Applications, LNCS 3485, pp. 79-93, 2005.
© Springer-Verlag Berlin Heidelberg 2005

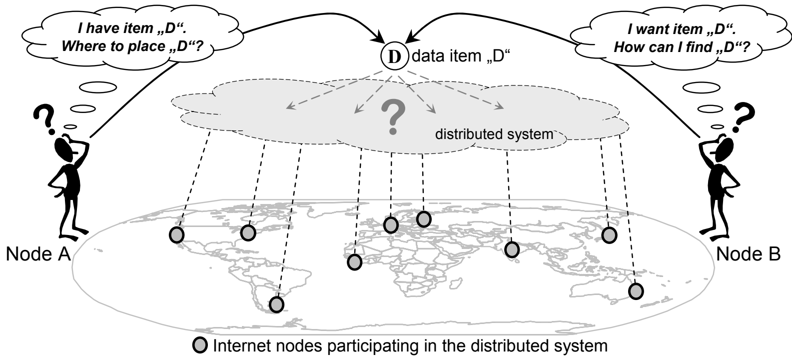


Fig. 7.1: The lookup problem: Node A wants to store a data item D in the distributed system. Node B wants to retrieve D without having prior knowledge of D's current location. How should the distributed system, especially data placement and retrieval, be organized (in particular, with regard to scalability and efficiency)?

7.1 Distributed Management and Retrieval of Data

Peer-to-Peer systems and applications raise many interesting research questions. Because of the completely decentralized character of Peer-to-Peer systems, the distributed coordination of resources, such as storage, computational power, human presence, and connectivity becomes a major challenge (Section 2.1). In most cases, these challenges can be reduced to a single problem: *Where to store, and how to find a certain data item in a distributed system without any centralized control or coordination?* (Figure 7.1) [52]

The lookup problem can be defined as follows: Some node A wants to store a data item D in the distributed system. D may be some (small) data item, the location of some bigger content, or coordination data, e.g., the current status of A, or its current IP address, etc. Then, we assume some node B wants to retrieve data item D later. The interesting questions are now:

- Where should node A store data item D?
- How do other nodes, e.g., node B, discover the location of D?
- How can the distributed system[1] be organized to assure scalability and efficiency?

The remainder of this section presents three approaches to answer these questions and discusses the advantages of drawbacks for each.

[1] In the context of Peer-to-Peer systems, the distributed system – the collection of participating nodes pursuing the same purpose – is often called the *overlay network* or *overlay system*.

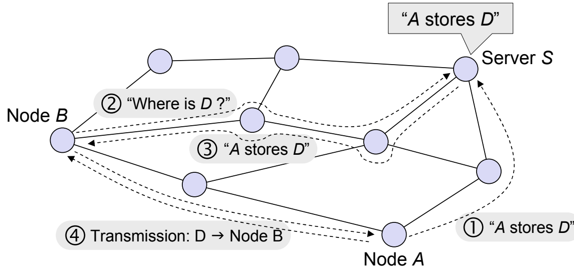


Fig. 7.2: Central Server: (1) Node A publishes its content on the central server S. (2) Some node B requests the actual location of a data item D from the central server S. (3) If existing, S replies with the actual location of D. (4) The requesting node B transmits the content directly from node A.

7.1.1 Comparison of Strategies for Data Retrieval

This section advocates the use of Distributed Hash Tables by comparing three basic strategies to store and retrieve data in distributed systems: *centralized servers*, *flooding search*, and *distributed indexing*.

7.1.2 Central Server

The approach of first generation Peer-to-Peer systems, such as Napster [436], is to maintain the current locations of data items in a central server. After joining the Peer-to-Peer system, a participating node submits to the central server information about the content it stores and/or the services it offers. Thus, requests are simply directed to the central server that responds to the requesting node with the current location of the data (Figure 7.2). Thereupon, the transmission of the located content is organized in a Peer-to-Peer fashion between the requesting node B and the node storing D.

The server-based approach is common in many application scenarios and was the major design principle for distributed applications in the past decades. It has the advantage of retrieving the location of the desired information with a search complexity of $O(1)$ – the requester just has to know the central server. Also, fuzzy and complex queries are possible, since the server has a global overview of all available content. However, the central server approach has major drawbacks, which have been become increasingly evident in recent years. The central server is a critical element within the whole system concerning scalability and availability. Since all location information is stored on a single machine, the complexity in terms of memory consumption is $O(N)$, with N representing the number of items available in the distributed system. The server also represents a single point of failure

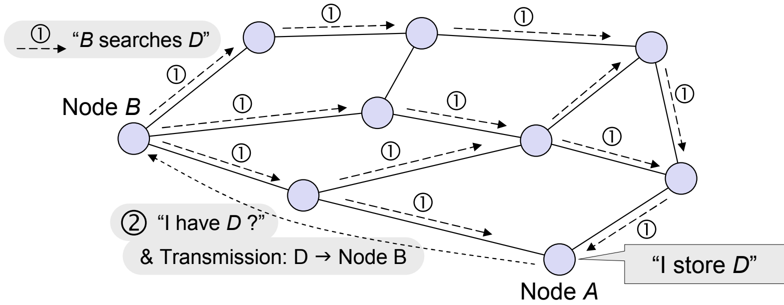


Fig. 7.3: Flooding Search: No routing information is maintained in intermediate nodes. (1) Node A sends a request for item D to its "neighbors" in the distributed system. They forward the request to further nodes in a recursive manner (flooding/breadth-first search). (2) Node(s) storing D send an answer to A, and A transmits D directly from the answering node(s).

and attack. If it fails or becomes unavailable for either of these reasons, the distributed system – as a whole – is no longer useable.

Overall, the central server approach is best for simple and small applications or systems with a limited number of participants, since the costs for data retrieval are in the order of $O(1)$ and the amount of network load (in proximity of the server) and the necessary storage capacity increase by $O(N)$. But, scalability and availability are vital properties, especially when systems grow by some orders of magnitude or when system availability is crucial. Therefore, more scalable and reliable solutions need to be investigated.

7.1.3 Flooding Search

Distributed systems with a central server are very vulnerable since all requests rely on the server's availability and consistency. An opposite approach is pursued by the so-called second generation of Peer-to-Peer systems (cf. Chapter 5.3). They keep no explicit information about the location of data items in other nodes, other than the nodes actually storing the content. This means that there is no additional information concerning where to find a specific item in the distributed system. Thus, to retrieve an item D the only chance is to ask as much participating nodes as necessary, whether or not they presently have item D, or not. Second generation Peer-to-Peer systems rely on this principle and broadcast a request for an item D among the nodes of the distributed system. If a node receives a query, it floods this message to other nodes until a certain hop count (Time to Live – TTL) is exceeded. Often, the general assumption is that content is replicated multiple times in the network, so a query may be answered in a small number of hops.

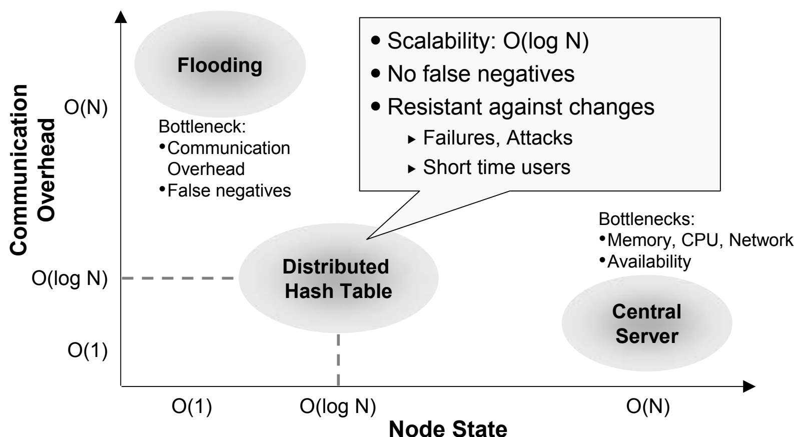


Fig. 7.4: Comparison of complexity in terms of search effort (y-axis) and storage cost per node (x-axis). Bottlenecks and special characteristics of each approach are named.

A well-known example of such an application is Gnutella [249], which is described in more detail in Section 3.4. Gnutella includes several mechanisms to avoid request loops, but it is obvious that such a broadcast mechanism does not scale well. The number of messages and the bandwidth consumed is extremely high and increases more than linearly with increasing numbers of participants. In fact, after the central server of Napster was shut down in July 2001 due to a court decision [438], an enormous number of Napster users migrated to the Gnutella network within a few days, and under this heavy network load the system collapsed (Section 3.4).

The advantage of flooding-based systems, such as Gnutella, is that there is no need for proactive efforts to maintain the network. Also, unsharp queries can be placed, and the nodes implicitly use proximity due to the expanding search mechanism. Furthermore, there is are efforts to be made when nodes join or leave the network.

But still, the complexity of looking up and retrieving a data item is $O(N^2)$, or even higher, and search results are not guaranteed, since the lifetime of request messages is restricted to a limited number of hops. On the other hand, storage cost is in the order of $O(1)$ because data is only stored in the nodes actually providing the data – whereby multiple sources are possible – and no information for a faster retrieval of data items is kept in intermediate nodes.

Overall, flooding search is an adequate technique for file-sharing-like purposes and complex queries.

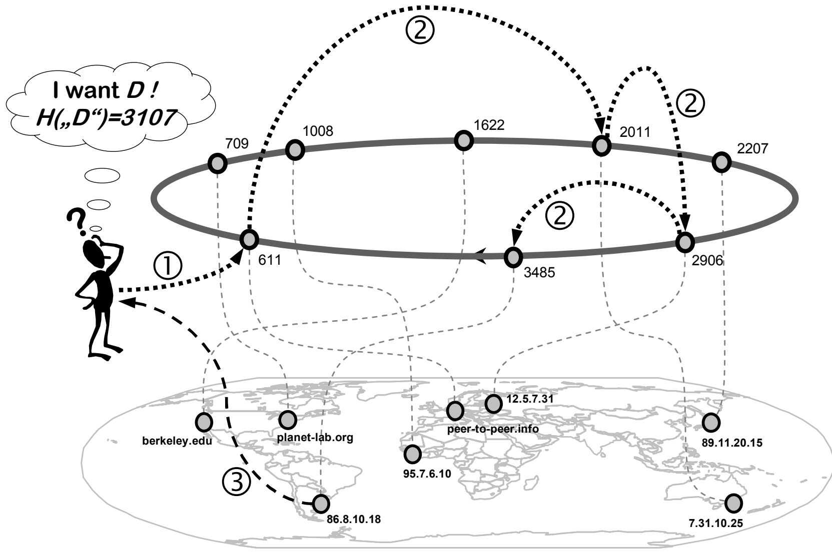


Fig. 7.5: Distributed Hash Table: The nodes in the distributed system organize themselves in a structured overlay and establish a small amount of routing information for quick and efficient routing to other overlay nodes. (1) Node A sends a request for item D to an arbitrary node of the DHT. (2) The request is forwarded according to DHT routing with $O(\log N)$ hops to the target node. (3) The target node sends D to node A.

7.1.4 Distributed Indexing – Distributed Hash Tables

Both central servers and flooding-based searching exibit crucial bottlenecks that contradict the targeted scalability and efficiency of Peer-to-Peer systems. Indeed, central servers disqualify themselves with a linear complexity for storage because they concentrate all references to data and nodes in one single system. Flooding-based approaches avoid the management of references on other nodes and, therefore, they require a costly breadth-first search which leads to scalability problems in terms of the communication overhead.

A better solution for the lookup problem should avoid these drawbacks and should enable scalability by finding the golden path between both approaches (Figure 7.4). In this case, scalability is defined as follows: the search and storage complexity per node should not increase significantly – by means not more than $O(\log N)$, even if the system grows by some orders of magnitude.

Distributed Indexing, most often in the form of Distributed Hash Tables, promises to be a suitable method for this purpose. In the realm of Peer-to-Peer systems, these approaches are also often called *structured Peer-to-Peer systems* because of their structured and proactive procedures. Distributed

Hash Tables provide a global view of data distributed among many nodes, independent of the actual location. Thereby, location of data depends on the current DHT state and not intrinsically on the data.

Overall, Distributed Hash Tables posses the following characteristics:

- In contrast to unstructured Peer-to-Peer systems, each DHT node manages a small number of references to other nodes. By means these are $O(\log N)$ references, where N depicts the number of nodes in the system.
- By mapping nodes and data items into a common address space, routing to a node leads to the data items for which a certain node is responsible.
- Queries are routed via a small number of nodes to the target node. Because of the small set of references each node manages, a data item can be located by routing via $O(\log N)$ hops. The initial node of a lookup request may be any node of the DHT.
- By distributing the identifiers of nodes and data items nearly equally throughout the system, the load for retrieving items should be balanced equally among all nodes.
- Because no node plays a distinct role within the system, the formation of hot spots or bottlenecks can be avoided. Also, the departure or dedicated elimination of a node should have no considerable effects on the functionality of a DHT. Therefore, Distributed Hash Tables are considered to be very robust against random failures and attacks.
- A distributed index provides a definitive answer about results. If a data item is stored in the system, the DHT guarantees that the data is found.

7.1.5 Comparison of Lookup Concepts

The following table compares again the main characteristics of the presented approaches in terms of complexity, vulnerability and query ability. According to their complexity in terms of communication overhead, per node state maintenance, and their resilience, Distributed Hash Tables show the best performance unless complex queries are not vital. For fuzzy or complex query patterns, unstructured Peer-to-Peer systems are still the best option.

System	Per Node State	Communication Overhead	Fuzzy Queries	Robustness
Central Server	$O(N)$	$O(1)$	✓	×
Flooding Search	$O(1)$	$\geq O(N^2)$	✓	✓
Distributed Hash Table	$O(\log N)$	$O(\log N)$	×	✓

Table 7.1: Comparison of central server, flooding search, and distributed indexing.

7.2 Fundamentals of Distributed Hash Tables

This section introduces the fundamentals of Distributed Hash Tables, such as data management, principles of routing, and maintenance mechanisms. Chapter 8 provides a detailed explanation of several selected Distributed Hash Table approaches.

7.2.1 Distributed Management of Data

A Distributed Hash Table manages data by distributing it across a number of nodes and implementing a routing scheme which allows one to efficiently look up the node on which a specific data item is located. In contrast to flooding-based searches in unstructured systems, each node in a DHT becomes responsible for a particular range of data items. Also, each node stores a partial view of the whole distributed system which effectively distributes the routing information. Based on this information, the routing procedure typically traverses several nodes, getting closer to the destination with each hop, until the destination node is reached.

Thus, Distributed Hash Tables follow a proactive strategy for data retrieval by structuring the search space and providing a deterministic routing scheme. In comparison, the routing information in unstructured systems is not related to the location of specific data items but only reflects connections between nodes. This reactive strategy results in queries being flooded on demand throughout the network because routing cannot be directed towards the lookup target. With a centralized system, the lookup strategy is implicit: routing a query (above the IP level) is unnecessary since the lookup procedure itself is confined to a single system.

7.2.2 Addressing in Distributed Hash Tables

Distributed Hash Tables introduce new address spaces into which data is mapped. Address spaces typically consist of large integer values, e.g., the range from 0 to $2^{160}-1$. Distributed Hash Tables achieve distributed indexing by assigning a contiguous portion of the address space to each participating node (Figure 7.6). Given a value from the address space, the main operation provided by a DHT system is the lookup function, i.e., to determine the node responsible for this value.

Distributed Hash Table approaches differ mainly in how they internally manage and partition their address space. In most cases, these schemes lend themselves to geometric interpretations of address spaces. As a simple example, all mathematical operations on the address space could be performed modulo its number of elements, yielding a ring-like topology.

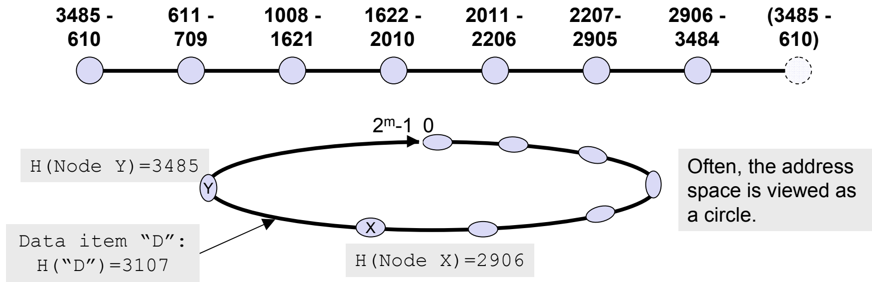


Fig. 7.6: A linear address space with integer values ranging from 0 to 65,535. The address space is partitioned among eight peers.

In a DHT system, each data item is assigned an identifier *ID*, a unique value from the address space. This value can be chosen freely by the application, but it is often derived from the data itself via a collision-resistant hash function, such as SHA-1 [207]. For example, the ID of a file could be the result of hashing the file name or the complete binary file. Thus, the DHT would store the file at the node responsible for the portion of the address space which contains the identifier.

The application interfaces of Distributed Hash Tables abstract from these details and provide simple but generic operations. Based on the lookup function, most DHTs also implement a storage interface similar to a hash table. Thus, the put function accepts an identifier and arbitrary data (e.g., the hash value of a file and the file contents) to store the data (on the node responsible for the ID). This identifier and the data is often referred to as *(key,value)*-tuple. Symmetrically, the get function retrieves the data associated with a specified identifier.

With this generic interface and the simple addressing scheme, Distributed Hash Tables can be used for a wide variety of applications. Applications are free to associate arbitrary semantics with identifiers, e.g., hashes of search keywords, database indexes, geographic coordinates, hierarchical directory-like binary names, etc. Thus, such diverse applications as distributed file systems, distributed databases, and routing systems have been developed on top of DHTs (see Chapters 11 and 12) [526, 367, 574, 373].

Most DHT systems attempt to spread the load of routing messages and of storing data on the participating nodes evenly (Chapter 9) [513, 450]. However, there are at least three reasons why some nodes in the system may experience higher loads than others: a node manages a very large portion of the address space, a node is responsible for a portion of the address space with a very large number of data items, or a node manages data items which are particularly popular. Under these circumstances, additional load-balancing mechanisms can help to spread the load more evenly over all nodes. For

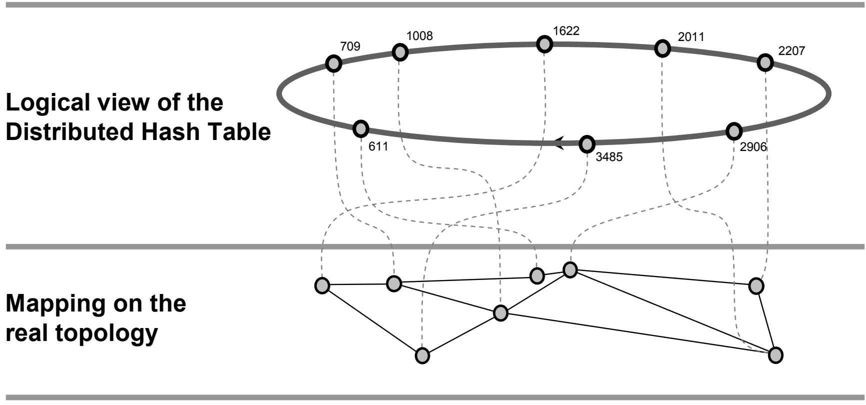


Fig. 7.7: Overall and underlay view of a Distributed Hash Table.

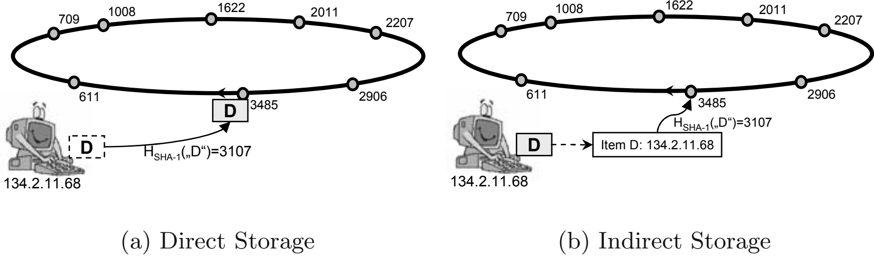
example, a node may transfer responsibility for a part of its address space to other nodes, or several nodes may manage the same portion of address space. Chapter 9 discusses load-balancing schemes in more detail.

7.2.3 Routing

Routing is a core functionality of Distributed Hash Tables. Based on a routing procedure, messages with their destination IDs are delivered to the DHT node which manages the destination ID. Thus, it is the routing algorithms of DHTs which solve the lookup problem.

Existing DHT systems implement a large variety of approaches to routing. However, the fundamental principle is to provide each node with a limited view of the whole system by storing on it a bounded number of links to other nodes. When a node receives a message for a destination ID it is not responsible for itself, it forwards the message to one of these other nodes. This process is repeated recursively until the destination node is found.

The choice of the next-hop node is determined by the routing algorithm and the routing metric. A typical metric is that of numeric closeness: messages are always forwarded to the node managing the identifiers numerically closest to the destination ID of the message. Ideally, such a scheme reliably routes a message to its destination in a small number of hops. Obviously, it is challenging to design routing algorithms and metrics such that node failures and incorrect routing information have limited or little impact on routing correctness and system stability.



(a) Direct Storage (b) Indirect Storage

Fig. 7.8: Two methods of storing data in Distributed Hash Tables.

7.2.4 Data Storage

There are two possibilities for storing data in a Distributed Hash Table. In a Distributed Hash Table which uses direct storage, the data is copied upon insertion to the node responsible for it (Figure 7.8(a)). The advantage is that the data is located directly in the Peer-to-Peer system and the node which inserted it can subsequently leave the DHT without the data becoming unavailable. The disadvantage is the overhead in terms of storage and network bandwidth. Since nodes may fail, the data must be replicated to several nodes to increase its availability. Additionally, for large data, a huge amount of storage is necessary on every node.

The other possibility is to store references to the data. The inserting node only places a pointer to the data into the Distributed Hash Table. The data itself remains on this node, leading to reduced load in the DHT (Figure 7.8(a)). However, the data is only available as long as the node is available.

In both cases, the node using the Distributed Hash Table for lookup purposes does not have to be part of the Distributed Hash Table in order to use its services. This allows to realize a DHT service as third-party infrastructure service, such as the OpenDHT Project [511].

7.3 DHT Mechanisms

Storage and retrieval of distributed data is the main purpose of Distributed Hash Tables. In this section, common mechanisms for the management of data and nodes in Distributed Hash Tables are discussed. These tasks address the insertion and retrieval of data and the arrival, departure, and failure of nodes.

7.3.1 Overview

To store or access data in a Distributed Hash Table, a node first needs to join it. The arrival of new nodes leads to changes in the DHT infrastructure, to which the routing information and distribution of data needs to be adapted. At this stage, the new node can insert data items into the Distributed Hash Table and retrieve data from it. In case a node fails or leaves the system, the DHT needs to detect and adapt to this situation.

7.3.2 Node Arrival

It takes four steps for a node to join a Distributed Hash Table. First, the new node has to get in contact with the Distributed Hash Table. Hence, with some bootstrap method it gets to know some arbitrary node of the DHT. This node is used as an entry point to the DHT until the new node is an equivalent member of the DHT. Then, the new node needs to be assigned a partition in the logical address space. Depending on the DHT implementation, a node may choose arbitrary or specific partitions on its own or it determines one based on the current state of the system. Third, the routing information in the system needs to be updated to reflect the presence of the new node. Fourth, the new node retrieves all $(key, value)$ pairs under its responsibility from the node that stored them previously.

7.3.3 Node Failure

Node failures must be assumed to occur frequently in distributed systems consisting of many unreliable and often poorly connected desktop machines. Thus, all non-local operations in a Distributed Hash Table need to resist failures of other nodes. This reflects the self-organizing design of DHT algorithms. They have to be designed to always fulfill their purpose and deal with all likely events and disruptions that may happen.

For example, routing and lookup procedures are typically designed to use alternative routes towards the destination when a failed node is encountered on the default route. This is an example of reactive recovery, i.e., a fault is handled during a regular DHT operation. Many Distributed Hash Tables also feature proactive recovery mechanisms, e.g., to maintain their routing information. Consequently, they periodically probe other nodes to check whether these nodes are still operational. If they are not, the corresponding routing entry is replaced with a working node.

Furthermore, node failures lead to a re-partitioning of the DHT's address space. This may in turn require *(key, value)*-pairs to be moved between nodes and additional maintenance operations such as adaptation to

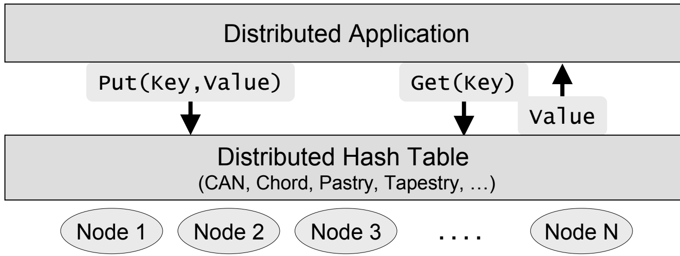


Fig. 7.9: Interface of a Distributed Hash Table. With a simple `put`-/textsfget-interface, the DHT simply abstracts from the distribution of data among nodes.

new load-balancing requirements. When a node fails, the application data that it stored is lost unless the Distributed Hash Table uses replication to keep multiple copies on different nodes. Some Distributed Hash Tables follow the simpler soft-state approach which does not guarantee persistence of data. Data items are pruned from the Distributed Hash Table unless the application refreshes them periodically. Therefore, a node failure leads to a temporary loss of application data until the data is refreshed.

7.3.4 Node Departure

In principle, nodes which voluntarily leave a Distributed Hash Table could be treated the same as failed nodes. However, DHT implementations often require departing nodes to notify the system before leaving. This allows other nodes to copy application data from the leaving node and to immediately update their routing information leading to improved routing efficiency. When triggered explicitly, replication and load-balancing mechanisms can also work more efficiently and reliably.

7.4 DHT Interfaces

There are two angles from which the functionality of Distributed Hash Tables can be viewed: they can be interpreted as routing systems or as storage systems. The first interpretation focuses on the delivery of packets to nodes in a DHT based on a destination ID. In the second, a Distributed Hash Table appears as a storage system similar to a hash table. These notions are reflected in the interface that a Distributed Hash Table provides to applications.

7.4.1 Routing Interface

Routing in a Distributed Hash Table is performed in the logical address space which is partitioned among the participating nodes. Any identifier from the address space can serve as a destination address for a message. Thus, the functionality provided by the DHT is to forward a message for an ID to the node which is responsible for this identifier.

An interface with two primitives suffices to build distributed applications on this foundation. The send primitive accepts a destination ID and a message and delivers the message from an arbitrary node in the system to the node which manages the destination ID. The receive primitive passes incoming messages and their destination identifiers to the application on the receiving node.

All other details of DHT management, such as node arrival and departure or repair mechanisms, are implemented by the Distributed Hash Table itself and are not exposed to the application. This generic, stateless interface implements little functionality but leaves a lot of flexibility to the application design. In particular, the storage and retrieval of data including load-balancing strategies can be implemented on top of the routing interface.

7.4.2 Storage Interface

As a storage system, a Distributed Hash Table implements an interface for persistently storing and reliably retrieving data in a distributed manner. On each node, the application interface provides the two main primitives of a hash table. The put primitive takes a $(key, value)$ pair and stores it on the node responsible for the identifier key. Similarly, the get primitive accepts an identifier and returns the value associated with the specified identifier.

The implementation of this interface adds to a Distributed Hash Table another level of complexity beyond correct and efficient routing. The storage layer needs to deal with routing failures, prevent data loss from node failure through replication, achieve load-balancing, provide accounting and admission control, etc. DHT implementations use different solutions to address these problems as described in Chapter 8.

7.4.3 Client Interface

Given the above interfaces, a node can only utilize its primitives after joining a Distributed Hash Table. However, a distributed system can also be structured such that the nodes participating in the DHT make available the DHT services to other, non-participating hosts. In such an environment, these hosts act as clients of the DHT nodes. This setup can be desirable where, for example, the Distributed Hash Table is run as an infrastructure service on a dedicated set of nodes for increased reliability. The interface between clients

and DHT nodes is also well-suited to realize access control and accounting for services available on the Distributed Hash Table. Note that this interface can itself be implemented as an application on top of the DHT routing or storage layer.

7.5 Conclusions

Distributed Hash Tables provide an efficient layer of abstraction for routing and managing data in distributed systems. By spreading routing information and data across multiple nodes, the scalability issues of centralized systems are avoided while data retrieval is significantly more efficient than in unstructured Peer-to-Peer networks. Also, the generic interface of Distributed Hash Tables supports a wide spectrum of applications and uses.

DHT implementations, such as those discussed in Chapter 8, focus on different conceptual and functional aspects. This is reflected in their different properties, such as scalability, routing latency, fault tolerance, and adaptability. Since the design of a Distributed Hash Table has to meet several, often conflicting, goals, each system exhibits its own strengths and weaknesses in different application scenarios.

Among these design challenges are:

– *Routing efficiency:* The latency of routing and lookup operations is influenced by the topology of the address space, the routing algorithm, the number of references to other nodes, the awareness of the IP-level topology, etc.
– *Management overhead:* The costs of maintaining the Distributed Hash Table under no load depend on such factors as the number of entries in routing tables, the number of links to other nodes, and the protocols for detecting failures.
– *Dynamics:* A large number of nodes joining and leaving a Distributed Hash Table – often referred to as *"churn"* – concurrently puts particular stress on the overall stability of the system, reducing routing efficiency, incurring additional management traffic, or even resulting in partitioned or defective systems.

Distributed Hash Tables also face fundamental challenges related to the principle of distributed indexing. For example, it is not clear how Distributed Hash Tables can operate reliably in an untrusted environment with Byzantine faults, i.e., when participating nodes are non-cooperative or malicious and damage the system. Furthermore, data retrieval in Distributed Hash Tables is based on numeric identifiers. Thus, query metrics based on tokens of strings or any other arbitrary data as well as fuzzy searches are very difficult to achieve efficiently. Among others, it is these challenges that will drive research in the area of Distributed Hash Tables in the future.