Distributed Systems
Project Report

Team 15

1 Distributed Hash Tables (DHT)

A **hash table**, also known as a **hash map** or a **hash set**, is a data structure that implements an associative array, also called a dictionary, which is an abstract data type that maps keys to values. A hash table uses a hash function to compute an *index*, also called a *hash code*, into an array of *buckets* or *slots*, from which the desired value can be found. During lookup, the key is hashed, and

the resulting hash indicates where the corresponding value is stored.

A distributed hash table (DHT) is a system that provides a lookup service like a hash table. Key–value pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. The main advantage of a DHT is that nodes can be added or removed with minimum work around re-distributing keys. *Keys* are unique identifiers which map to values, which in turn can be anything from addresses, to documents, to arbitrary data. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. DHTs form an infrastructure that can be used to build more complex services, such as anycast, cooperative web caching, distributed file systems, domain name services, instant messaging, multicast, peer-to-peer file sharing and content distribution systems.

1.1 DHT Properties

DHTs characteristically emphasize the following properties:

- **Autonomy and decentralization**: The nodes collectively form the system without any central coordination.
- **Fault tolerance**: The system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.
- Scalability: The system should function efficiently even with thousands or millions of nodes.

A key technique used to achieve these goals is that any one node needs to coordinate with only a few other nodes in the system – most commonly, O(log *n*) of the *n* participants so that only a limited amount of work needs to be done for each change in membership.

1.2 DHT Structure

The structure of a DHT can be decomposed into several main components. The foundation is an abstract keyspace, such as the set of 160-bit strings. A keyspace partitioning scheme splits ownership of this keyspace among the participating nodes. An overlay network then connects the nodes, allowing them to find the owner of any given key in the keyspace.

1.2.1 Keyspace Partitioning

Most DHTs use some variant of consistent hashing or rendezvous hashing to map keys to nodes.

1.2.2 Overlay Network

Each node maintains a set of links to other nodes (its *neighbors* or routing table). Together, these links form the overlay network.

2 DHT using Chord

The Chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure (or planned departure) of existing nodes. Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. It uses consistent hashing. Chord improves

the scalability of consistent hashing by avoiding the requirement that every node knows about every other node.

A Chord node needs only a small amount of "routing" information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with a few other nodes. In a N-node network, each node maintains information only about O(log N) other nodes, and a lookup requires O(log N) messages. Chord must update the routing information when a node joins or leaves the network; a join or leave requires $O(\log^2 N)$ messages.

2.1 Consistent Hashing

The consistent hash function assigns each node and key an m-bit identifier using a base hash function such as SHA-1. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. The identifier length must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Identifiers are ordered in an identifier circle modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the successor node of key k, denoted by successor(k). If identifiers are represented as a circle of numbers from 0 to $2^m - 1$, then successor(k) is the first node clockwise from k.

2.2 Scalable Key Location

A very small amount of routing information suffices to implement consistent hashing in a distributed environment. Each node need only be aware of its successor node on the circle. Queries for a given identifier can be passed around the circle via these successor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to.

A portion of the Chord protocol maintains these successor pointers, thus ensuring that all lookups are resolved correctly. This resolution scheme is inefficient: it may require traversing all N nodes to find the appropriate mapping. To accelerate this process, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as the successor information is maintained correctly.

Let m be the number of bits in the key/node identifiers. Each node, n, maintains a routing table with (at most) entries, called the **finger table**. The i^{th} entry in the table at node n contains the identity of the first node that succeeds n by at least 2^{i-1} on the identifier circle. This is called the **finger** of the node n. A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node.

2.3 Node Joins

In a dynamic network, nodes can join (and leave) at any time. The main challenge in implementing these operations is preserving the

ability to locate every key in the network. To achieve this goal, Chord needs to preserve two invariants:

- Each node's successor is correctly maintained.
- For every key k, node successor(k) is responsible for k.

Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node n joins the network, certain keys previously assigned to n's successor now become assigned to n. When node n leaves the network, all of its assigned keys are reassigned to n's successor. No other changes in assignment of keys to nodes need occur.

To simplify the join and leave mechanisms, each node in Chord maintains a predecessor pointer. A node's predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node and can be used to walk counterclockwise around the identifier circle.

To preserve the invariants stated above, Chord must perform three tasks when a node n joins the network:

- Initialize the predecessor and fingers of node n.
- Update the fingers and predecessors of existing nodes to reflect the addition of n.
- Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node is now responsible for.

2.4 Node Failure

When a node n fails, nodes whose finger tables include n must find n's successor. In addition, the failure of n must not be allowed to disrupt queries that are in progress as the system is retabilizing. The key step in failure recovery is maintaining correct successor pointers, since in the worst case find predecessor can make progress using only successors. After a node failure, other nodes may attempt to send requests through the failed node as part of a find successor lookup. Ideally the lookups would be able to proceed, after a timeout, by another path despite the failure. All that is needed is a list of alternate nodes, easily found in the finger table entries preceding that of the failed node. If the failed node had a very low finger table index, nodes in the successor-list are also available as alternates.