An Overview of Distributed Hash Tables (DHT)

Vaibhav Thakkar
Department of Electrical Engineering, Indian Institute of Technology Kanpur, Uttar Pradesh, 208016
Email: vaithak@iitk.ac.in

***Abstract*—This tutorial paper first gives a brief introduction about hash tables and it's terminologies and concepts, then it discusses about distributed hash tables and especially about it's very famous key partitioning scheme - *consistent hashing*. We will also discuss the use case of this distributed key-value service and how peer to peer services like Bit Torrent protocol uses it effectively.**

I. Introduction

A hash table is an efficient data structure that implements an associative array abstract data type which provides a fast lookup of arbitrary data using a unique key. An entry in hash table is written notationally as a (key, value) pair, where key is the unique identifier and value is the actual data associated with that key which will be actually stored in the table.

Hash tables offer a combination of efficient lookup, insert and delete operations of a (key, value) pair in constant time with very high probability.

A distributed hash table is used for providing a scalable key-value lookup service in which any node participating in the network can retrieve the key:value pair efficiently. Keys are the unique identifiers on the basis of which searching happens in a hash table (not necessarily distributed), keys map to particular values, which in turn can be anything for example addresses, actual documents or even arbitrary data. Responsibility for maintaining the mapping from keys to values or the hash table is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption (atleast that's what the aim is).

A. Overview of a hash table

There are two steps involved in a general hash table implementation, let's say our aim is to insert (same steps for delete and lookup) a (key, value) pair into our hash table arbitrary data into our hash table.

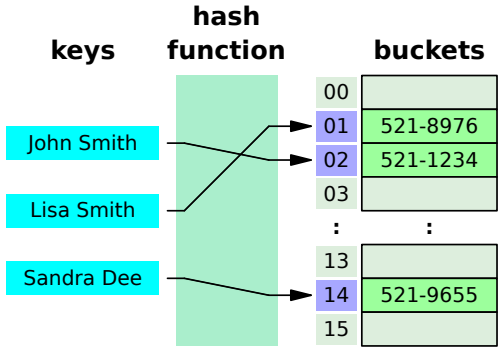

Fig. 1. Overview of Hash Tables

1) Pre-Hashing: Firstly there is a "pre-hash" stage which is used to map keys to integers, this step is necessary as not only the values we are inserting in the hash table can be arbitrary but also the key used for looking that value can be anything like strings or urls. Consider an example with which we interact with every time we browse the internet, the DNS service or protocol which is used for mapping URL's which are a human friendly way of interacting with internet to IP addresses which are machine or specifically computer friendly way of interacting with internet, this protocol definitely requires a fast way to lookup an IP Address using the URL provided, so in this case the key used for lookup is actually a URL which is a string. When most people talk about hashing this is the step they are talking about which one may argue is not actually the right term as this is actually pre-hashing but still this is prevalent. There are several algorithms which have been developed and improved over a course of many years that are used for this step, some of the most common of which are MD5, SHA1, SHA2, SHA3 etc. one thing to note that MD5 hash function is not cryptographically secure and the algorithm used for breaking it (meaning calculating what input produced that output from hash function) can be easily run within few minutes on a modern day computer [1]. One can even create one's own hash function by analysing the type of keys required to be hashed.

2) Hashing: Actually the origin of word hashing is a verb from French 'hache' meaning hatchet. The use of this step is to reduce universe U of all keys (say, integers) down to reasonable size m for table.

Thus we can define the hash function as,

$$h : U \to \{0, 1, \cdots, m-1\}$$

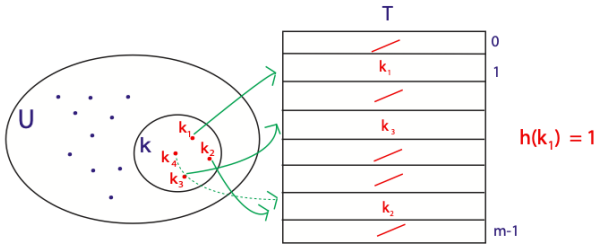

Fig. 2. Mapping keys to hash table

One important thing to note that for hash table to work correctly the function in both the steps should be deterministic and have the property that if same input is passed many times same output is obtained, this condition is very important as if this is not satisfied you may not be able to search your (key, value) pair you inserted.

Now we will introduce one more problem which is faced in implementing hash tables knowns as **Collisions**, as the name suggests this problem occurs when two distinct keys map to same location in our hash table. With basic mathematical reasoning of functions one can easily verify that this condition can't be removed completely with any hash function, but there are techniques like Chaining and Probing to handle collisions such that the overall time complexity of the hash table operations are not effected very much asymptotically.

A critical statistic for a hash table is the load factor,

$$load\ factor = \frac{n}{k}$$

where
n is the number of entries occupied in the hash table
k is the number of buckets.
As the load factor grows larger, the hash table becomes slower. The expected constant time property of a hash table assumes that the load factor be kept below some bound.

B. Anatomy of Distributed Hash Tables

Now let's get started with distributed hash tables. DHT research was originally motivated, in part, by peer-to-peer systems which took advantage of resources (like increased network capacity and storage) distributed across the Internet to provide a single useful application.

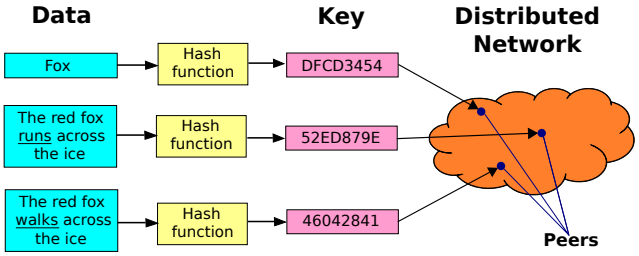

Fig. 3. Distributed Hash Tables

1) Structure of a DHT: The structure of a DHT can be decomposed into several main components [2]. One of the main components is the **keyspace** such as the set of 160-bit strings. Notes that this keyspace is the universe of keys generated by the hash function in pre-hash step.

A keyspace partitioning scheme is responsible for splitting the ownership of this keyspace among the participating nodes.

An overlay network is the one that connects the nodes so that they communicate with each other to allow finding the node on which a given key is present. Each node maintains a routing table containing links to other nodes. Together, these links form the overlay network. A node picks its neighbours according to the network's topology.
Let's see how all these things come together to allowing efficient indexing and retieval of arbitrary data:-

A typical workflow of the DHT for storage and retrieval might proceed as follows. Suppose the keyspace is the set of 160-bit strings (which is the common keyspace used by many hash functions). To index a file with given filename and data in the DHT, the SHA-1 hash of filename is generated, producing a 160-bit key k, and a message put(k, data) is sent to any node participating in the DHT, now these nodes will communicate in the overlay network to finally take the data to the node responsible for handling this key as specified by the keyspace partitioning scheme. That node now stores the key and the data in it's internal hash table. Any other client can then retrieve the contents of the file by again hashing filename to produce k and asking any DHT node to find the data associated with k with a message get(k). The message will again be routed through the overlay to the node responsible for k, which will reply with the stored data.
Some of the the common DHT designs are Tapestry [3], Chord [4], Kademlia [5], Pastry [6] and many more. These systems follow the *consistent hashing* scheme. Now we will discuss key partition schemes namely **Consistent Hashning** [7] and **Rendezvous Hashing**, but before that we will also discuss a toy key partitioning scheme which will be based on modulo operator.

II. Key Partitioning Schemes

A. Modulo Key Partitioning Scheme

This scheme tries to distribute the load evenly between nodes Let's say we have n nodes in the network, to distribute keys to the nodes we divide the keys such that a node has to store on average only $\frac{1}{n}$ of the keys,
we do so by assigning key k to the $(hash(k)\ mod\ n)$ numbered node, where nodes are number from $0, 1, \cdots (n-1)$.
This scheme works well if the number of nodes or serves participating in the network remain fixed but one of the characteristic properties of the DHT's are that they are *fault tolerant*, meaning DHT's should be reliable even with nodes continuously joining, leaving or failing.
In this scheme even if one node fails meaning if the n changes, there will be movement of data in all the nodes thus making it slow.This scheme is very easy to implement but it has a major disadvantage: *The Rehashing problem.*
Can we even avoid the rehashing problem ?
No, but we can reduce the number of nodes affected by the rehashing. This brings us to our next partitioning scheme.

B. Consistent Hashing

The previous partitioning scheme used number theoretic approach for distributing keys, this scheme uses another geometry and distance metrics to partition the keys among the participating nodes. The main idea of this scheme is that nodes and keys map to the same space. In this scheme each node is assigned a single number called its identifier (ID). A node with ID i_x owns all the keys k_m for which i_x is the closest ID, measured according to a distance metric $\delta(k_m, i_x)$.
For example, the Chord DHT uses consistent hashing, which treats nodes as points on a circle, and $\delta(k_1, k_2)$ is the distance traveling clockwise around the circle from k_1 to k_2.
Thus, each key is first mapped onto the circle and then the circular key space is split into contiguous segments whose endpoints are the node identifiers.
This can be implemented by another way by mapping node identifiers as well as keys onto the number line from $[0, 1]$ (which is equivalent of unfolding the circle in 1 dimension), and

mapping a key to the node on it's right.

The figure below illustrates the scheme in a better way.

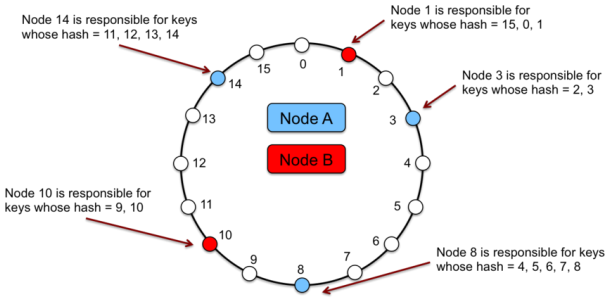

Fig. 4. Consistent hashing scheme

Consistent hashing is a special kind of hashing such that when a hash table is resized, only $\frac{K}{n}$ keys need to be remapped on average, where
K is the number of keys, and n is the number of slots.
Thus, when a node leaves only those keys which were stored by the leaving node would have to be rehashed to the next node in clockwise order.
Similarly, if a new node enters only some of the keys that were initially stored by the node's current clockwise neighbour will be rehashed and stored in the new node.
The task of finding the clockwise neighbour can be implemented by maintaining a Balanced Binary Search Tree of all the nodes in the network such that comparison between nodes is done using their identifiers and for finding the neighbour one can find the node's predecessor or successor in the binary search tree.
This implementation is easy to implement and memory efficient as generally the number of nodes in the network are very small as compared to the number of keys.
Let's compare asymptotic time complexities for N nodes (or slots) and K keys.

Operation	Modulo Partitioning	Consistent Hashing
add a node	$O(K)$	$O(\frac{K}{N} + log(n))$
remove a node	$O(K)$	$O(\frac{K}{N} + log(n))$
add a key	$O(1)$	$O(log(n))$
remove a key	$O(1)$	$O(log(n))$

Now we will prove a lemma for obtaining a bound on load factor for a node in this scheme. We will assume that the the nodes are assigned ID's randomly and independently from, also we will assume that the keys to be inserted are also random.

Lemma 1. *With high probability no node owns more than $O(\frac{ln(n)}{n})$ fraction of the total circle, where n are the number of nodes.*

Proof. We will consider the interval $[0, 1]$, all points on the circle can also mapped to this interval by using the function $\frac{\theta}{2\pi}$ where θ is the angle in radians.
Now divide the interval into intervals of length $\frac{2*ln(n)}{n}$

Consider any interval, the probability that none of the node lands in it

$$= (1 - \frac{2ln(n)}{n})^n \quad (1)$$

$$\leq \frac{1}{n^2} \quad (2)$$

Equation 2 can be derived as follows.
We know that

$$(1 - x) \leq e^{-x}$$

$$\Rightarrow (1 - \frac{2ln(n)}{n})^n \leq (e^{-ln(n^2)/n})^n$$

$$= (e^{ln(\frac{1}{n^2})} = \frac{1}{n^2}$$

Now we will use the Union bound theorem which says that

$$\mathbb{P}\left(\bigcup_i A_i\right) \leq \sum_i \mathbb{P}(A_i).$$

$$\Rightarrow \mathbb{P}(\text{There exists an interval such that no machine lands in it}) \leq n \times \frac{1}{n^2} = \frac{1}{n}$$

Applying Union bound on intervals, we get
With probability $\geq (1 - \frac{1}{n})$, each machine owns at most $\frac{4ln(n)}{n}$ of the circle. □

C. Rendezvous Hashing

We will not discuss this scheme thoroughly but just introduce it and get a basic idea of what it does.
In rendezvous hashing[1], all the nodes or clients use the same hash function h (chosen ahead of time) to associate a key to one of the n available nodes.
For this, each client should have the same list of identifiers $\{S_1, S_2, \cdots, S_n\}$, one for each server.
Given some key k, a client computes n hash weights

$$w_1 = h(S_1, k)$$
$$w_2 = h(S_2, k)$$
$$\cdots$$
$$w_n = h(S_n, k)$$

The client associates that key with the server corresponding to the highest hash weight for that key. Thus we can say that the server S_x owns all the keys k_m for which the hash weight $h(S_x, k_m)$ is higher than the hash weight of any other node for that key.
One can see that *consistent hashing* is a special case for *rendezvous hashing* as the delta function can be used to assign the hash weights as

$$h(S_x, k) = -\delta(k, id(S_x))$$

[1]also called highest random weight hashing

III. Summary

We discussed hash tables and how they can run in distributed way, thus offering an efficient (key, value) insertion and retrieval service which can handle large amounts of data by utilizing the resources of all participating nodes or systems in the network. We discussed the key partitioning scheme - *consistent hashing* which helps in minimizing the overheads of entering or leaving of a node from the network. We also discussed how *BitTorrent* protocol uses Distributed hash table as a core part of it's peer to peer file transfer implementation.

References

[1] Y. Sasaki and K. Aoki, "Finding preimages in full md5 faster than exhaustive search," in *Advances in Cryptology - EUROCRYPT 2009* (A. Joux, ed.), (Berlin, Heidelberg), pp. 134–152, Springer Berlin Heidelberg, 2009.

[2] M. Naor and U. Wieder, "Novel architectures for p2p applications: the continuous-discrete approach," 2006.

[3] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, EECS Department, University of California, Berkeley, Apr 2001.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, Aug. 2001.

[5] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems* (P. Druschel, F. Kaashoek, and A. Rowstron, eds.), (Berlin, Heidelberg), pp. 53–65, Springer Berlin Heidelberg, 2002.

[6] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001* (R. Guerraoui, ed.), (Berlin, Heidelberg), pp. 329–350, Springer Berlin Heidelberg, 2001.

[7] D. Karger, E. Lehman, F. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web.," pp. 654–663, 01 1997.