An Overview of Distributed Hash Tables (DHT)

Vaibhav Thakkar
Undergraduate, Electrical Engineering, Indian Institute of Technology Kanpur, Uttar Pradesh
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 prove some lemmas related to bounds on the load factors of the consistent hashing scheme. 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 a data structure which is used for providing an efficient indexing and 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, keys map to particular values, which on the other hand can be any arbitrary data. The ownership for maintaining the key to value mappings in the hash table is distributed among all the nodes participating in the network, this is done while ensuring that any change in the set of participants causes very little disruption.

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.

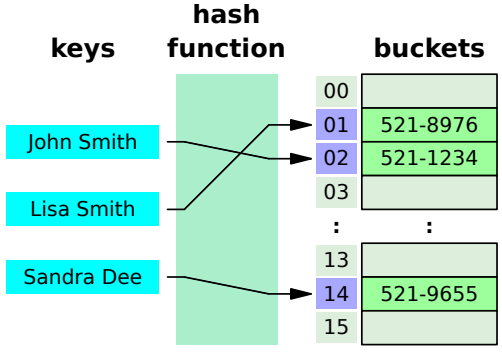

Fig. 1. Overview of Hash Tables

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,

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

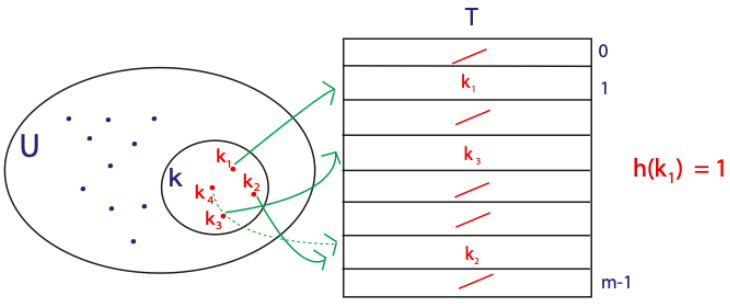

Fig. 2. Mapping keys to hash table

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.

An important statistic used for comparing various implementation of a hash table is the load factor,

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

where
n = count of (key, value) pairs present in hash table
k = count of number of buckets.
With increasing load factor, the hash table will become slower. The efficiency of hash table, i.e 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. The research on Distributed Hash Table was motivated by peer to peer systems which took advantage of resources (like increased network capacity and storage) that were distributed across the network to provide one useful application.

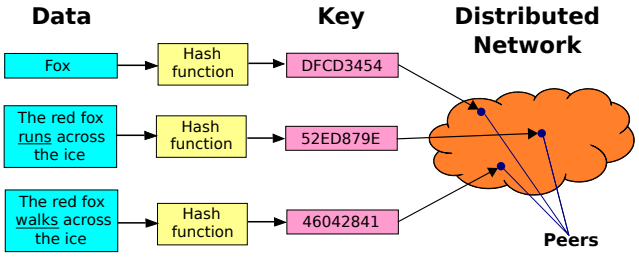

Fig. 3. Distributed Hash Tables

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 partitioning the ownership or storage of this key-space 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:-

Let's walk through how in a Distributed Hash Table storage and retrieval operations might proceed. Suppose that the key space produced after the pre-hashing step is the set of 160-bit strings. To insert or store a particular file with a given file name and data in the Distributed Hash Table, the SHA-1 hash of the provided lookup key which is the file name in this case is generated, producing a 160 bit key k which is then passed to the next step, by a message INSERT(k, data) which is sent to any node in the overlay network of 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 partitioning scheme. That node now stores the key and the data in it's internal hash table. Now for retrieval of that data a client can hash the file name again to produce k and can ask any node in DHT to find the data associated with k with a message RETRIEVE(k). The message will again be routed through the network to the node which owns the key k, which will reply with the stored file data.
Some of the the common DHT designs are Tapestry [3], Chord [4], Kademlia [5], Pastry [6] and there are several others. All the above examples 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 our overlay network of the DHT, 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*.
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 partitioned into contiguous intervals with node IDs being there endpoints.
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.

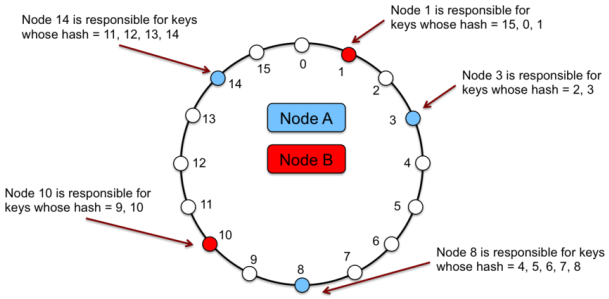

Fig. 4. Consistent hashing scheme

Consistent hashing is a type of implementing hashing which ensures when a hash table is resized, on average only $\frac{K}{n}$ keys need to be moved, where
K = the count of keys present in the table, and n = the total count of slots or max keys that can be stores.
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 and K keys.

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

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

$$\leq \frac{1}{n^2} \tag{2}$$

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.

□

After obtaining the upper bound, one can also think about the lower bound, i.e how small of a fraction of the circle can be between two nodes,
which is equivalent to saying how close can two nodes be on the circle.
Intuitively, one can think that the lower bound can be very small, which is indeed true. We won't be calculating the exact lower bound but we will see that the lower bound is even less than $\frac{1}{n^2}$.
That means, there is a good probability that the fraction of circle between two machines is even less than $\frac{1}{n^2}$
We can prove this by again mapping the circle onto $[0, 1]$ interval, then we will divide the interval into intervals of length $\frac{1}{n^2}$,
Now using the Birthday paradox, we know that the probability that there exist atleast 2 nodes out of n nodes that lie in the same interval $\geq \frac{1}{2}$.
Thus, the probability that the smallest fraction of circle between any two nodes in the network of a Distributed Hash Table is $\geq \frac{1}{2}$.

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 are provided with the same hash function h (which is decided beforehand) to associate a key to one of the n nodes.
For every node to able to perform this step, each node should have the same list of identifiers $\{S_1, S_2, \cdots, S_n\}$, one for each server.
Now for any given key k, a node computes all of the n hash weights for that key

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

The node associates that key with the node corresponding whose identifier produced the highest hash weight for that particular key. Thus any node S_x owns all the keys k_m for which it has the highest hash weight $h(S_x, k_m)$ for that key among all the nodes.
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

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

III. Application: How BitTorrent uses DHT

From Wikipedia [8] *"The BitTorrent protocol is used to reduce the server and network impact of distributing large files. Rather than downloading a file from a single source server, the BitTorrent protocol allows users to join a "swarm" of hosts to upload to/download from each other simultaneously"*.
BitTorrent protocol is a widely used peer to peer file transfer protocol. In this paper we will assume you are aware of the terminology used in BitTorrent protocol like peers, seeds, torrent files and trackers. One of the most essential part of this protocol is peer discovery. It is one of the main reasons how the downloads happen so quickly: You connect to many peers, and then each of them is uploading a little piece of the file to you and you (may) do the same for them. Now, to connect to the peers you have to know first enter into the BitTorrent network and you just can't broadcast your IP on the internet because Internet works in the unicast manner unlike the local subnets. This phase of entering into the BitTorrent network by connecting to some of the peers is called the *Bootstrapping* phase, but how do you connect to even one peer ? That's where trackers comes in, each tracker stores a list of peers which are exchanging a particular torrent file, thus maintaining a hash table with the unique key as the torrent file's infohash.
So, in order to bootstrap into the network a client has to connect to a tracker which will then connect it to the required peers.

[1]also called highest random weight hashing

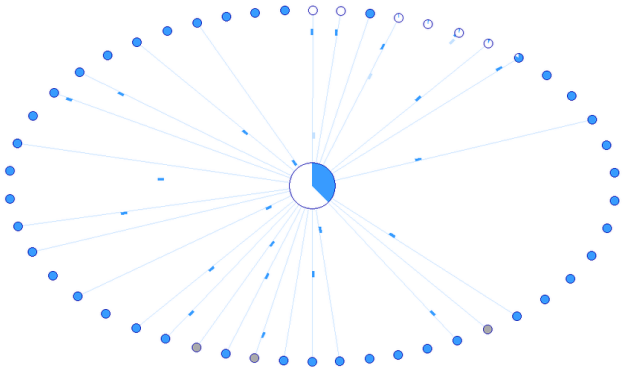

Fig. 5. Downloading file from peers

Now to make torrent files trackerless, BitTorrent uses DHT for storing peer contact information. Due to this, effectively each peer becomes a tracker as each Each BitTorrent client is also a DHT node. The design of DHT used in BitTorrent is based on Kademlia [5] and is implemented over the UDP protocol. But the question that comes is how do you even connect to a DHT node without knowing any IP address, the answer is *you can't*, each BitTorrent client is already hard coded to some specific static IP or DNS using which you connect to a stable DHT node, which provides the DHT metadata for connecting further into the network. So strictly speaking, the BitTorrent DHT is not decentralised completely, as for bootstrapping it depends on some fixed number of nodes.
Kademlia uses consistent hashing's aprroach, thus each node is assigned an identifier which is unique globally known as the "node ID" which are chosen randomly and independently from the same 160 bit space as BitTorrent infohashes. The "distance metric" used in kademlia for comparing a node ID with another node ID or an infohash for "closeness" is XOR and the result is stored as an unsigned integer.

$$distance(A, B) = |A \ xor \ B|$$

Kademlia uses a trie data structure to use the geometric properties of xor to find the neighbouring nodes or infohashes, which helps in finding neighbours in logn time.

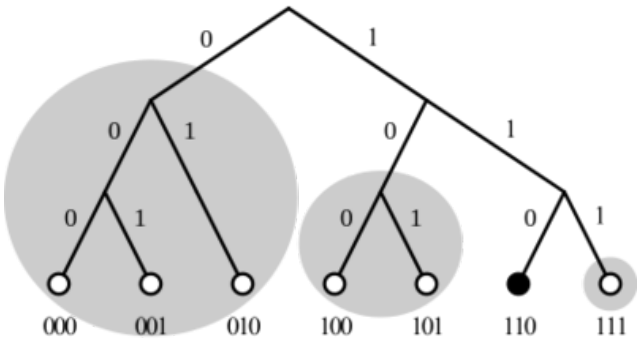

Fig. 6. Kademlia Trie made using the XOR metric

Each node maintains a routing table which contains the information necessary for communicating with other nodes. The routing table gets more and more detailed as the stored node IDs get closer to the node's own ID.
Any node contains information about a lot of nodes in the DHT that have their identifiers which are near (according to the distance metric) to their own ID but contains info about comparatively less number of nodes which have with IDs that are far away (accorsing to distance metric) from their own. ID
When a node wants to find peer of a particular infohash it sends a query to the nodes which are present in it's routing table and are closest to that infohash according to the above distance metric. It then communicates with those nodes asking for the information about the peers that are downloading that torrent file, the response for which can either be the peer information or the list of nodes that are closes to the torrent file's infohash the other responding node's routing table. The starting node keeps on requesting for the peer information iteratively till it finds it or the search is exhausted, along with the process of finding peer information each node in this process also keeps on updating it's routing table with the information it retrieves about other nodes.
After finding the peers information, the client can download the data from all the peers bit by bit.

IV. 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. I have also coded an implementation of the Chord and Kademlia design of DHT in C++ which is available on https://github.com/vaithak/Distributed-Hash-Table

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.

[8] Wikipedia, "Bittorrent protocol."