Chord: Scalable Peer To Peer Lookup Protocol

Submitted by:

Gajanan Modi(2019201049)

Aditya Gupta (2019201067)

Amit Jindal (2019201037)

Anupam Misra (2019201082)

To:

Professor Lini Thomas

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

April 2021

Table of content

Abstract

A fundamental problem that confronts peer-to-peer applications is the efficient location of the node that stores a desired data item. This project presents Chord, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis and simulations show that Chord is scalable: communication cost and the state maintained by each node scale logarithmically with the number of Chord nodes.

1. Introduction

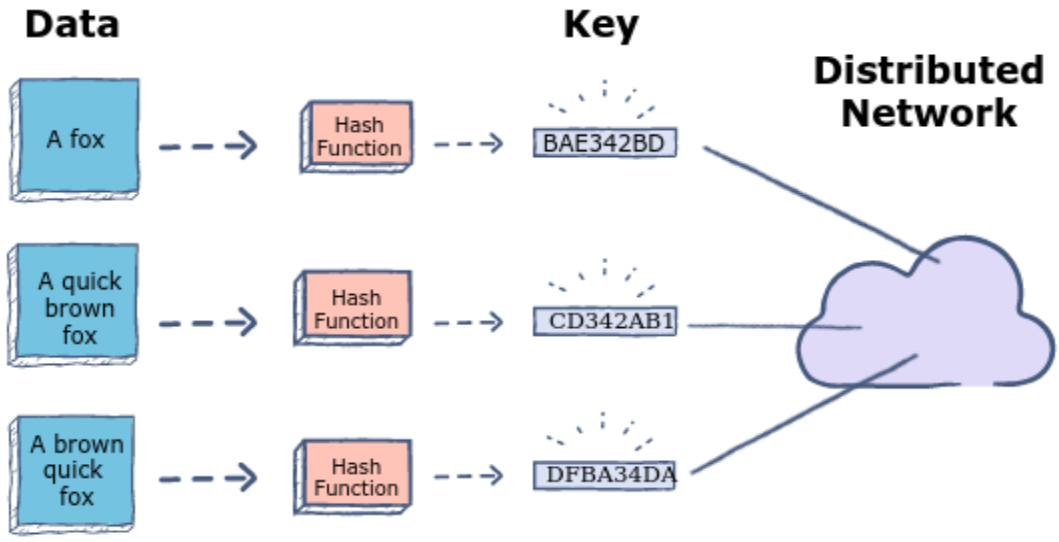
- We have implemented a **decentralized distributed hash table** using the **CHORD** protocol.
- The Chord protocol supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key
- Chord network adapt efficiently as nodes join and leave the system.
- It also supports **Fault tolerance** in case of node failure.
- It has **logarithmic worst-case time complexity** to search a value for a corresponding key.
- Chord uses **Consistent hashing** to make it more reliable and efficient.
- **Timely repair:** Each node will regularly ask for acknowledgement from its neighbours to know that they are still present in the network. If no acknowledgement is received then they have left the network and stabilization is done accordingly.

2. Theoretical bases and literature review

1. **Distributed Hash Table**

- A distributed hash table (DHT) is a decentralized storage system that provides lookup and storage schemes similar to a hash table, storing key-value pairs.
- Each node in a DHT is responsible for keys along with the mapped values. Any node can efficiently retrieve the value associated with a given key.

- Just like in hash tables, values mapped against keys in a DHT can be any arbitrary form of data.



DHTs have the following properties:

- ***Decentralised & Autonomous:*** Nodes collectively form the system without any central authority.
- ***Fault Tolerant:*** System is reliable with lots of nodes joining, leaving, and failing at all times.
- ***Scalable:*** System should function efficiently with even thousands or millions of nodes.

Just like hash tables, DHTs support the following 2 functions:

1. put (key, value)
2. get (key)

2. Consistent Hashing

- Consistent Hashing is a special kind of hashing such that when a hash table is resized, only n/m keys need to be remapped on average where n is the number of keys and m is the number of slots
- The consistent hash function assigns each node and key an m-bit identifier using SHA-1 as a base hash function.
- 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 m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

- Consistent hashing assigns keys to nodes as follows:
  Identifiers are ordered on an identifier circle (known as Chord ring) 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.
- **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.

3. Algorithm and Design

1. Topological Structure of Chord
   - is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.The Chord protocol supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses consistent hashing to assign keys to Chord nodes.
   - the Chord library provides a lookup(key) function that yields the IP address of the node responsible for the key. Second, the Chord software on each node notifies the application of changes in the set of keys that the node is responsible for.

2. Scalable Key Location
   - To accelerate lookups, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as each node knows its correct successor.
   - A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Note that the first finger of n is the immediate successor of n on the circle; for convenience we often refer to the first finger as the successor.
   - This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Second, a node's finger table generally does not contain enough information to directly determine the successor of an arbitrary key k.

3. Impact of Node Joins on Lookup
    - we consider the impact of node joins on lookups. We first consider correctness.

    - If joining nodes affect some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors. The common case is that all the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor in O(logN) steps. The second case is where successor pointers are correct, but fingers are inaccurate.

    - This yields correct lookups, but they may be slower. In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail. The higher-layer software using Chord will notice that the desired data was not found,

4. Failure and Replication
    - The correctness of the Chord protocol relies on the fact that each node knows its successor.

    - To increase robustness, each Chord node maintains a successor list of size r, containing the node's first r successors. If a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All r successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of r.

    - A modified version of the closest preceding node procedure searches not only the finger table but also the successor list for the most immediate predecessor of id. In addition, the pseudocode needs to be enhanced to handle node failures.If a node fails during the find successor procedure, the lookup proceeds, after a timeout, by trying the next best predecessor among the nodes in the finger table and the successor list. The following results quantify the robustness of the Chord protocol, by showing that neither the

4. Commands for running/functions

- Each machine that will act as a Chord node needs to run
  `./ChordClient specify-some-port-number-here`
  `eg.: ./ChordClient 12345`

- To create a new Chord ring:
  `Create_chord`

- To join a Chord ring created by another node (NOTE: Can specify the IP and port of ANY node which is a part of that Chord ring, not necessarily the creator):
  `join_chord node-ip-address node-port-number`

- To check who is(are) the current node's successor(s):
  `Display_successor_list`

- To check who is the current node's predecessor:
  `Display_predecessor`

- To check the current node's identifier:
  `Display_node_identifier`

- To view the current node's finger table:
  `Display_finger_table`

- To insert a new key value pair into the chord ring:
  `insert_key put-some-string-key put-some-string-value`

- To display the entries in the current node's hash table:
  `Display_hash_table`

- To search for a key in the chord ring (and retrieve its value, from whichever node has it):
  `search_key put-key-string-here`

- To leave the chord ring and exit the program:
  `leave_chord`

5. Conclusion

The Chord protocol solves the problem of determining the node that stores a data item in distributed peer to peer applications in a decentralized manner. It offers a powerful primitive: given a key, it determines the node responsible for storing the key's value, and does so efficiently. In the steady state, in an N -node network, each node maintains routing information for only O(log N) other nodes, and resolves all lookups via O(log N) messages to other nodes.

Attractive features of Chord include its simplicity, provable correctness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, albeit at degraded performance, when a node's information is only partially correct. It is also that Chord scales well with the number of nodes, recovers from large numbers

of simultaneous node failures and joins, and answers most lookups correctly even during recovery.

Chord has widespread applications for peer-to-peer, large-scale distributed applications such as cooperative file sharing, time-shared available storage systems, distributed indices for document and service discovery, and large-scale distributed computing platforms.

6. Video Link

https://iiitaphyd-my.sharepoint.com/:v:/g/personal/gajanan_modi_students_iiit_ac_in/EZutBr7I_QZAg0kcl8T1TOUBqPlmZhh6EnwAO5UL-eAKHg?e=DRCufR

7.References

All references are present in zip folder