Distributed Hash Table Design Document

XUZHENG LU

G34363475

ZETIAN ZHENG

G36558368

XIAOYU SHEN

G33150730

GUANGYUAN SHEN

G48268794

1 INTRODUCTION

A distributed hash table (DHT) is a protocol for structured peer-to-peer (p2p) networks used. Each node in the network follows a specific rule to divide and store the large file index hash table. The DHT entries should be the form: (key, value). The key is the hash value of the file, and the value is the IP address where the file is stored. It is efficient to find and return value/address from the query node. (Wikipedia: peer to peer)

In this distributed hash table, the joined nodes will automatically form a decentralized structure without the scheduling from the master node. In this way, there will be no system crash when the master node is failed or busy. Also, when a large number of nodes are flooded into the system, there may be a short-term load imbalance. The system will randomly allocate the same amount of storage to each node. Since the hash table can have node exit or sudden node failure, we also need to address the issue of exiting the nodes to ensure query and data stability.

2 ARCHITECTURE & ALGORITHM

2.1 Main idea

The main idea of DHT are as follows:

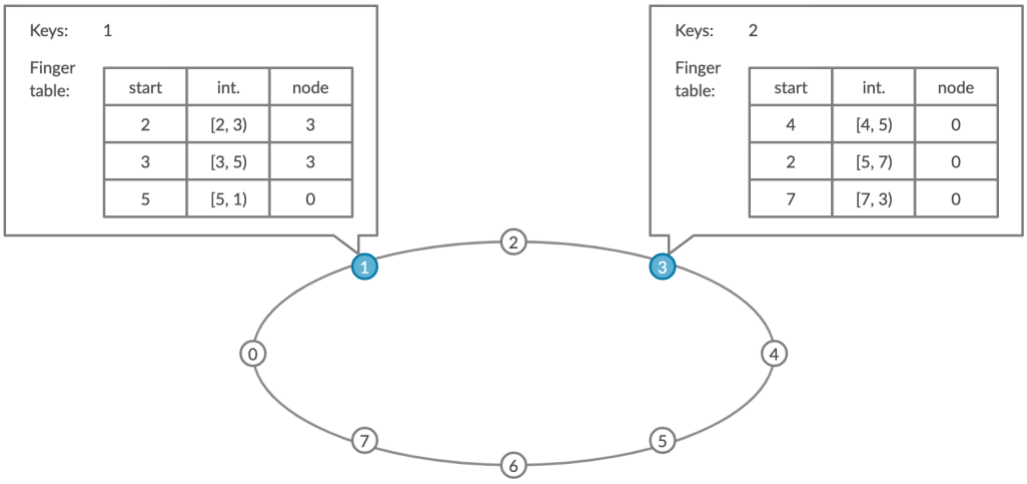
1. Each file index is represented as a (K, V) pair. K is the keyword, which can be the hash value of the file name (or other information of the file), and V is the IP address of the node that stores the file (or other information of the node).
2. All the file index entries (i.e. (K, V) pairs) form a large hash table. We can find the node address that stores a file as long as we input the K value.
3. The large file hash table will be divided into many local small blocks. The local hash tables are distributed to all nodes in the system according to specific rules so that each node is responsible for maintaining one of the blocks.
4. When a node queries a file, it only needs to route the query message to the corresponding node, where the hash table block maintained the (K, V) pair to be searched.

2.2 Architecture

We will use the architecture similar to Chord DHT proposed by Ion Stoica et al. in 2003 (Stoica et al., 2003). Chord covers the entire network by stringing the nodes' IDs from small to large to form a ring, and each data is stored at the backward node closest to the data key. Chord uses the similar dichotomy search method, a non-

linear search algorithm, and the convergence speed can be very fast. The routing complexity can be reduced to the logarithm of the number of nodes.

The structure of a basic Chord-based system is shown as follows:



1. **NID (Node Identifier):** an m-bits number representing a physical machine. m must be large enough to make the probability of sharing the same NID be negligible.
2. **KID (Key Identifier):** an m-bits number representing a key (or resource). Key is hash-bound to a resource so that the KID can be obtained by the Key through the hash operation. m must be large enough to make the probability of sharing the same NID be negligible.
3. **Constant hash function:** Compared with the general hash function, the joining and leaving of nodes have the least impact on the entire system using the constant hash function. SHA-1 is used in Chord for constant hash calculation.
4. **Chord Ring:** NID and KID are allocated on a ring with the size of 2^m, for key distribution, node distribution, and key location. The key is allocated to the node with NID>=KID. This node becomes the successor node of k, the first node in the clockwise direction from k on the ring, denoted as successor(k). The nodes are placed on the ring from large to small in the clockwise direction.

2.3 Query (Key Location)

The key location is the core function of the Chord system. When node N is looking for a key with KID=id, it will ask whether the key is on the next node. If the KID is between the NID of the node and the NID of the next node, then the key is allocated to the next node. Otherwise, the same query will be asked on the next node, recursively. Finally, the location of the key will be found.

The query time complexity is O(N), where N is the number of nodes in the ring. To accelerate the searching process, Chord uses finger tables to maintain up to m entries on each node, where m is the number of bits in the hash key.

2.4 Add New Node

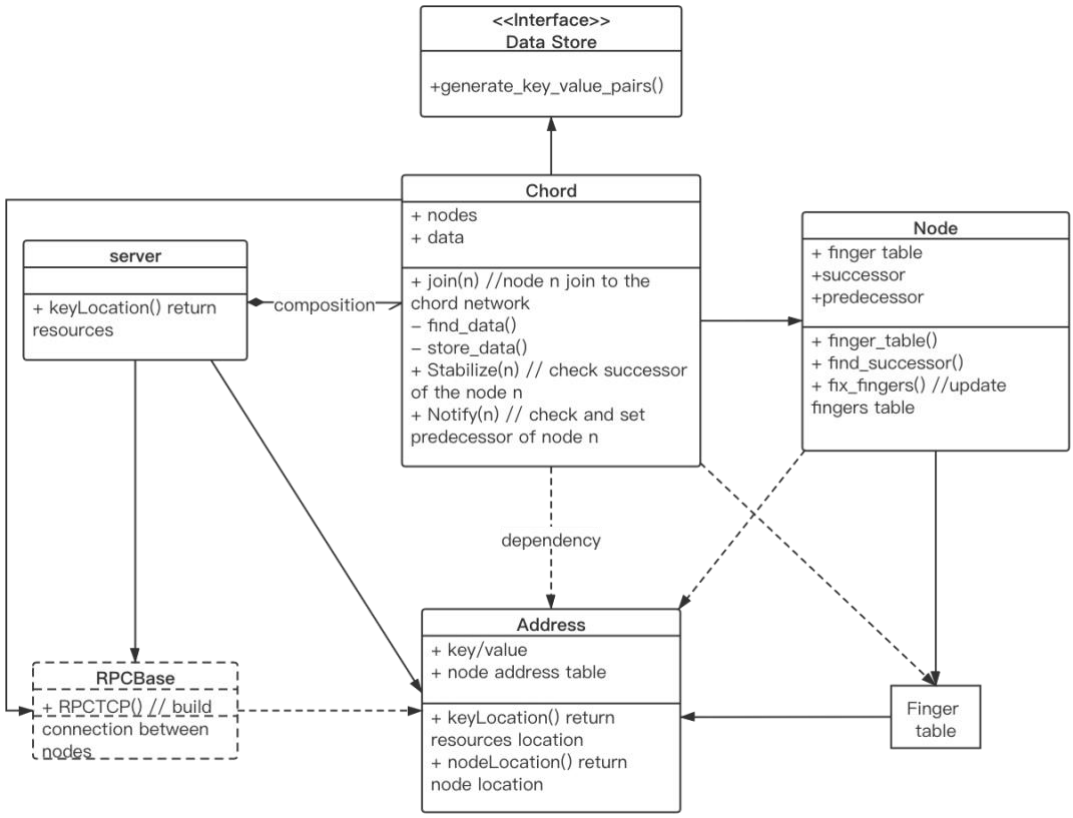
When there is a new node added into the system, we must maintain three invariants:

1. The successors of every node should point to their successors correctly.
2. Every key should be correctly stored in the successor(k).

3. The finger tables on every node should be updated correctly.

3 IMPLEMENTATION

The following UML diagram shows the high-level structure of the implementation of the project.



According to the implementation and performance of Chord (Kasperi Kaija, 2018), the chord system consists of several parts:

1. The core class Chord is the main control object which follows the chord protocol. Chord can find or store the data by interface Datastore, controlling nodes join in, and stabilize the whole chord ring(update the successor and predecessor of each node).
2. The server interface object is used to find the specific resources' key location and associate to object RPCBase, which uses TCP/IP protocols to build connections between nodes.
3. The Node class object is to track every node. It can find the finger table, successor, and predecessor of each node and fix the finger table after new nodes join.
4. The Address object responds to the request from the Server, Chord, and Node. It can return KID(key identifiers) and NID(node identifiers) by hash calculation.

4 CHALLENGES

There are several challenges in implementing a distributed hash table, such as load balancing, fault tolerance, and scalability.

4.1 Load Balance

As nodes and data continue to join the network, there will be situations where some nodes have a lot of data, and others have very little. Therefore, we need some way to distribute the data relatively evenly in the network so that when a node exits or suddenly fails, we will not lose or migrate much data.

Base on David R. Karger and Matthias Ruhl's work (Karger et al., 2004), we will take address space balancing. We will start by dividing the responsible space of each node equally into 1/n. If we use traditional random allocation, the average amount allocated per node will be log n. In the distributed hash table for the implementation of the Chord algorithm, we also need to consider the possibility that when a node is inserted or exited, it may cause a change in the routing table or the data location needs to be moved. Repeatedly querying or deleting data from the same node can cause the data to move repeatedly. So, we need to reduce the risk of repeated movement, so here we consider adding an appropriate number of virtual nodes to minimize the unnecessary overhead.

In addition to ensuring a random mapping of the address space, we also need to consider that certain data needs to be mapped to specified space. Where traditional distributed hash tables restrict the movement of data, we will choose to leave the restriction on movement open so that the data from the nodes can be moved at will. Since the node data can flow freely, and each node has a maximum space limit, we can balance the data within the network to a suitable level by simply moving it.

4.2 Fault Tolerance

Node exits or sudden errors are common in distributed hash tables, so we need to consider what we should do to ensure data integrity when this happens.

According to Moni Naor and Udi Wieder's work (Naor el al., 2003), to ensure fault tolerance, we have to make sure that the node's routing table is up to date, and if I access a non-existent or error node, it will be a system crash. This is to ensure that the front and back nodes of each node are online by exchanging information and updating the routing table information if there is an error on the front and back nodes.

4.3 Scalability

As nodes are added, the entire distributed hash table becomes extremely large, and we have to consider how to reorganize the hash table so that the efficiency will not drop.

When a new node is added, the node is still constantly checking the node before and after, updating the routing table if it is a new node, and keeping it unchanged if it is the original node. Besides, other nodes need to be notified of the addition of new nodes. This is also a dynamic adjustment of the load on each node so that the load balancing.

5 SCADULE

5.1 Tasks

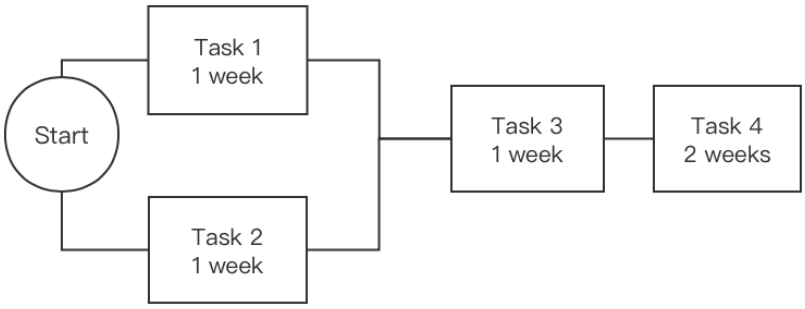
We split the project into five small tasks

1. **Task 1: Basic query**
   This task includes some basic functions to realize the query function like the "get" of the HashMap in java.
2. **Task 2: Finger Table**
   This task is an implementation of the figure table, which includes designing a new data struct for figure table and basic functions for it.
3. **Task 3: Add New Node**

This task is meant for implementing a new data struct for Nodes and the related changes when a new node is added into the chord.

4. **Task 4: System stability**
   - To make sure that each query is correct, update the successor and predecessor of each node.
   - Consider more factors that may influence the system like memory usage, CPU and network and so on. Try to enhance the robustness of the system and shorten the response time.
   - Tests and debug.

The basic schedule of these tasks is shown in the figure below:



5.2 Process Checking

We plan to have a regular meeting every Friday night to check the processing. But we will track and manage our work every day. And We design a table for recording the weekly task. Here is a simple template.

	Task	Work Log
Week 1	Task 1	
	Task 2	
Week 2	Task 2	
	Task 3	
Week 3	Taks 3	
Week 4	Task 4	

5.3 Issues Recording

We also create an issue table for recording our problem status. The table below is a simple prototype.

Issue ID	Date	Task	Editor	Problem	Solved or not
1	11/09	Task 1	Guangyuan Shen	Cannot build the findID function	yes
2	11/10	Task 1	Xiaoyu Shen	Solve Issue 1	-
3	11/12	Task 2	Zetian Zheng	Some test samples cannot pass	no
4	11/14	Task 1	XuZheng Lu	Update Issue 2	-

REFERENCES

[1] Kaija, Kasperi. "The implementation and performance of Chord." (2018).

[2] Stoica, Ion, et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications." IEEE/ACM Transactions on networking 11.1 (2003): 17-32.

[3] Karger, David R., and Matthias Ruhl. "Simple efficient load balancing algorithms for peer-to-peer systems." Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. 2004.

[4] Naor, Moni, and Udi Wieder. "A simple fault tolerant distributed hash table." International Workshop on Peer-to-Peer Systems. Springer, Berlin, Heidelberg, 2003.

[5] https://medium.com/techlog/chord-building-a-dht-distributed-hash-table-in-golang-67c3ce17417b

[6] https://www.cnblogs.com/gnuhpc/archive/2012/01/13/2321476.html