Chord Technical Report

Zhaowei Huang **Tianyi Wang**

1 Introduction

In this project, we implemented a key-value storage system based on Chord. Chord is a scalable peer-to-peer lookup protocol, which supports balanced load, fast lookup time(about O(logN)), server nodes join and leave. Chord uses consistent hashing algorithm that balance the distribution of keys on each node. Every Chord server is assigned to an address in a Chord ring, which has address space of 2^M, where M is the number of bits in the address locating. We set the address space of the system to be 2^{32}, which means there are at most 2^{32} nodes in the system. The selected hashing algorithm should evenly distribute these nodes to the ring such that every neighbor nodes should have similar distance. Chord also assigns each key-value pair an ID by applying the same consistent hashing algorithm to the key. Then, this pair is stored in the node whose node ID is closest to and greater than (or equal to) the key-value ID.

Chord distinguishes itself from other distributed key-value storage systems in two ways. First, the searching time of a key is log(N), where N is the number of server nodes in the ring. Therefore, millions of put/get requests can be accomplished in a short time. Second, since Chord is not a centralized distributed system and each node is a participant of the ring, it can easily scale up by accepting more nodes to join the ring, and it is not vulnerable to single point failure, as stored data can be redistributed to other alive nodes. This paper is organized in the following way. Section 2 provides more details of Chord logistics, and Section 3 introduces the architecture we build for simulating Chord algorithms. Finally, we provide our results of Chord experiments in Section 4.

2 More of Chord

2.1 Chord details

The log(N) time complexity of searching and putting data is achieved by a data structure named finger-table, which stores a list of target node address for some keys. Assuming the current node has id = I, and the system's address space is 2^M. The node addresses stored in the finger-table are $I + 2^0, I + 2^1, I + 2^2, ..., I + 2^{M-1}$. Finger table gets updated periodically based on information received from other nodes, like the successor/predecessor node. Since each key should be stored in the first server node that lies behind the key in the ring, the table updating process starts from the last target node address, using searching mechanism

such as binary search, the expected reduced space should always be 1/2 since the key and node are all generated by SHA-1 hashing. Finally, every node can find the correct target address for a key within log(N) talks to other nodes.

For the Chord algorithm, we must ensure the balanced distribution of keys, so we use SHA-1 as the hashing algorithm. For a chord server, we acquire its ID in the ring by applying SHA-1 to the concatenation of node IP address and port (IP:PORT),and for all key-value pairs, we directly apply SHA-1 on the key to generate the data ID, and store the key-value pair based on this data ID.

Each node would have a successor and predecessor, which are the node right behind it and the node right ahead of it in the ring. As long as all nodes record its genuine successor and predecessor nodes, all entries in its own finger-table finally should store correct information, since each new node can use its neighbors' correct finger-table to update its own, and by induction, the ring should become stable as every node holds a right finger table. From the perspective of implementation, These information can be updated and checked by routines like stabilize, check_predecessor and fix_fingers, which are called periodically. stabilize and check_predecessor ensures the node always has a correct successor and predecessor, and fix_fingers is used to update the finger-table.

2.2 Replication of data

Since running server may crash or get some unexpected error, it's important to replicate data in a distributed system. There are several ways to replicate data in the chord system.

2.2.1 Replicate on the ring

One mechanism is to replicate data on the ring. Each node in the system can replicate its data on its successor, so there would be a total of 2 replicate of each data. When a node N failed, its successor N_s has all N's data, so the data would not be lost. Also, the successor and predecessor of N should detect N's failure, and the predecessor N_p becomes the new predecessor of N_s, so it should replicate its data to N_s, which is originally replicated on N.

2.2.2 Use other replication algorithms on each node

Another way is to have a small cluster of nodes for each node in the ring. We can simply use some distributed consistent & fault tolerant algorithm on each node, like the Raft and Paxos algorithm. For example, each node in the ring could be a primary node in a raft cluster of size 3. If the node in the ring failed, which is the primary node in the raft cluster, a new leader would be elected and serve as the node in the ring. However, since the Chord algorithm use SHA-1 hashing to compute the id of node in the ring, the new primary should have the same key to be hashed, so we cannot use the ip-port pair to compute the node id here, since each node in a raft cluster would have different ip/port pair, and we want to ensure that each node in one cluster should have the same hashed id. We can apply some

other kinds of naming of the node, then simply ensure each replica of one cluster has the same key when it is used as the node in the chord ring.

2.3 Potential use of Chord

The feature of Chord is the O(logN) time complexity and balanced load of each server node in the ring, as well as the dynamically join/leave of node. Also, it provides a global searching even if the client just has a connect with one node in the ring. Therefore, Chord can be applied on some applications like P2P file transfer clients, distributed indices, time-shared storage system that share one node's data with others, etc.

3 Report Basics

3.1 Configuration

We tested the system on a single AWS m5.large instance. However, for this system, we didn't test its performance like the throughput/latency for two reasons. First, we are using a single AWS instance, and thus the latency of server communication is equivalent to inter process communication (IPC). Even if we decide to run multiple instances, the communication overhead is still negligible since these instances are in the same region under the same private networks. The second reason is that, due to hardware limitation, the maximum number of virtual CPU supported by a single instance is 2. Since we decide to run multiple virtual servers on the same instance, throughput is also restricted by hardware resource. Therefore, we focus on the balanced distribution and searching time, as the former measurement is not affected by hardware condition, and the latter can be achieved by metering path length of each operation, rather than absolute request latency.

3.2 System Structure & Implementation

In this system, we have three main modules: Chord server, Chord client, Statics manager. Chord server simulates a node in the ring, such that it uses the chord algorithm and stores the key-value pair that is uploaded by Chord client. Our Chord server implements all functions presented in the Chord paper, including $find\_successor$, $closest\_preceding\_node$, $create$, $join$, $stabilize$, $notify$, $fix\_fingers$, $check\_predecessor$.

We didn't implement virtual node in this experiment. We believe that virtual node is an excellent mechanism to balance the load, but implementing virtual node is not an easy work, and the testing would become more complicated in the experiment.

Besides, each Chord server has a one bit value $isAlive$, which indicates the status of the server. This value can be modified by the client by sending gRPC calls leaveRing or joinRing, which enforce the node to leave or join a ring. If $isAlive$ is set to false, that Chord server would not send/response any gRPC calls. Such implementation is a cheap way to simulate the condition of node failure or voluntary leave. We adopt this method to avoid

the inconvenience of using ctrl-C to kill the process. Also, this function-call mechanism can support more operations like redistributing data and notifying node's successor/predecessor etc.

Chord client is the module that operates on chord server, it supports sending several concurrent key-value pairs to servers, and call some operational gRPC on servers, like the leaveRing/joinRing gRPC. Also, it is able to get data from Statics Manager to help us get the statics of the system's behavior after storing a huge amount of key-value pairs.

Statics Manager is the module that stores the statics of the chord servers, such as the number of keys on each server node, and the path length count for all the data put operation. The path length of a put operation is incremented by one each time the server calls a get_successor gRPC on other server, and finally sent to the statics manager at the time the data is put on the target server node.

4 Experiment & Performance Test

The scenario of experiment is, we ran 16/32/64/128 server nodes in the chord ring, and put 10,000 random key-value pairs on the cluster in each experiment. We tried to run 1024 servers like the chord paper, but it seems impossible on the instance we use. This could be caused by some implementation defects and the limitation of the server hardware. By putting as many as 10,000 random key-value pairs to the ring, we assume that Ids generated by these keys could be ideally randomized, and if nodes are evenly distributed in the ring, all nodes should store the same amount of data. However, we believe that 16/32 servers cannot guarantee a balanced load in a M=32 chord ring, since it's highly possible that the server node itself is not evenly distributed on the ring, and the length between different pair of two consecutive nodes may vary, which makes the number of key-value pairs on each server is not balanced. However, for a ring with 64/128 servers, our experiment suggests that node distribution is relatively even, and the result of our trial is similar to the case with a maximum of 4096 servers from the original chord paper.

First, we record the number of keys on each server. If the number of server nodes is large enough, each node should store $\frac{K}{N}$ data. Next, we present the distribution of path length of 10,000 independent put operation. We would expect each put operation to have a maximum of O(logN) searching time, represented by path length. The largest length of path should be log(N), where N is the number of server nodes in the ring. For 16/32/64/128 server nodes, the maximum path should be 4/5/6/7.

4.1 Number of Data on Nodes

The probability density function (pdf) of number of keys on each nodes is shown in Figure 1. When there are 16 servers, nodes are not evenly distributed in the ring, so a single node may store more than 10 percent of total data. Most nodes store roughly 300 data, which is less than ideal $\frac{10000}{16} = 625$ data.

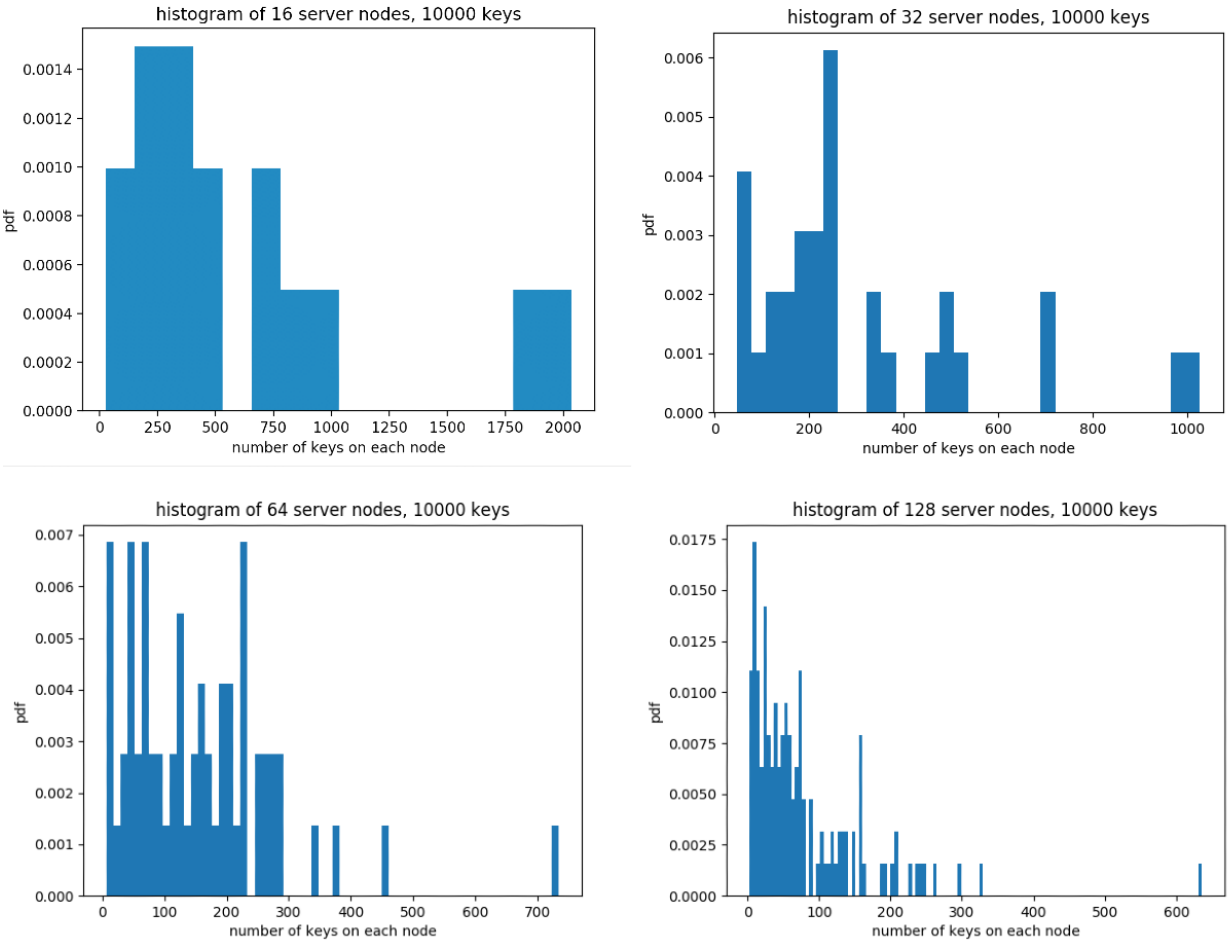

Figure 1: number of keys on node with 16/32/64/128 server nodes

When the server count becomes 32, we observe a better data distribution, shown in the upper right plot of Figure 1, as most nodes store about 250 key-value pairs, close to ideal $\frac{10000}{32} = 312$ data. However, the deviation of amount of data in each node is still large, and many servers only have less than 50 data. Once again, we believe the SHA-1 hashing algorithm could not evenly distribute 32 nodes to the ring as large as 2^{32} address spaces.

With node amount increased to 64, the most common data count is in the range [20, 220]. This range already suggests balanced data distribution, even though there still exists deviations for nodes storing more than 700 data pairs. Nevertheless, we still expect the range of most frequent data to be narrower.

Finally, when we activate 128 servers in a ring and perform 10,000 independent put operation, we obtain the graph in the lower right corner of Figure 1. As we can see, most

nodes store data within the range [10, 90]. This window is close and tight enough to indicate that we achieve the expectation of $\frac{10000}{128} = 78$ data per node when the server count is 128.

We expect that, if the number of server nodes is large enough, the distribution of number of keys on each node will be same as the presented result in the Chord paper, where the number of stored keys in most server nodes would be in range of $\frac{K}{2N}$ to $\frac{2K}{N}$. K is the total number of keys, and N is the number of server nodes. Even though, we can find this trend in our experiment result.

4.2 Path Length Distribution

We collect the path length distribution for 16/32/64/128 servers. As we expected, the maximum path length for each experiment are 4, 5, 6 and 7. Meanwhile, for each number of servers, we expect a Gaussian distribution of path length, such that most put operation requires log(N)/2 RPC calls to locate the right receiver, and zero or log(N) RPC calls should be rare. From our experiment, all server configurations (16/32/64/128) have such bell curve distribution. Figure 2 plots the path length distribution of 64 servers. Therefore, most put operations has time complexity of log(N)/2, for all N = 16, 32, 64 and 128. Having this observed, we notice that in the previous section, we assume 10,000 put operation has uniformly distributed data ID, but in reality, nodes might be evenly distributed around the ring, but the deviation of these data IDs could be an important factor of unbalanced storage distribution.

path length	0	1	2	3	4
count	635	3272	3908	1783	402

Table 1: path length distribution with 16 server nodes.

path length	0	1	2	3	4	5
count	325	1702	3719	2896	1202	156

Table 2: path length distribution with 32 server nodes.

path length	0	1	2	3	4	5	6
count	166	1016	2362	3415	2345	658	38

Table 3: path length distribution with 64 server nodes.

path length	0	1	2	3	4	5	6	7
count	72	530	1757	2908	2855	1532	325	21

Table 4: path length distribution with 128 server nodes.

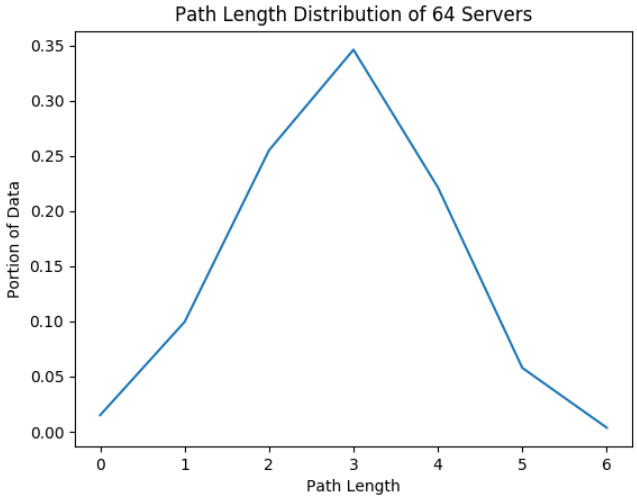

Figure 2: path length pdf of 64 servers

5 Conclusions

Chord protocol uses decentralized servers to store key-value pairs. Different from centralized application of distributed system, Chord has less concern of single point failure, since data are distributed across different nodes, and the protocol can still function properly even when some participants failed. In our implementation, Chord ring can stabilize in a short time when a single node joins or voluntarily leaves the ring. We have not yet implemented chain replication, and thus some data would loss in our system in a node exits the ring due to failure. However, fault tolerance is not our major objective in this paper, and we focus on the how efficient Chord is to search for a data, and how balanced data can be loaded to servers. Our experiment suggests that Chord can evenly push data to different servers, as long as the consistent hashing algorithm ensures high degree of randomness. Once the server count scales up to 128, our implementation can achieve an ideal load balancing among servers. Moreover, searching and putting data in Chord is very efficient. The complexity in the worst case is O(log(N)), but in normal case the operation can be done in O(log(N)/2). Therefore, the performance of Chord does not slow down even when the number of serves in the ring scales up. Scalibility is another reason we may prefer decentralized to centralized storage system.