Extended Chord Implementation with Data Replication and Manager Mode

Runze Xu, Xuhui Lu, Chuping Wang

***Abstract*—In this paper, we implemented a Chord-based distributed key-value store system and extended the original implementation with data replication and an alternative manager mode. Our data replication mechanism stores replicas on the successor list of the primary storage node. The successor list is now used to maintain both the Chord node ring's functionality and the number of replicas in the whole cluster. We also implemented a manager mode for the system. Although the manager is a single point of failure and bottleneck, we argue that it can help improve the system efficiency when the cluster load is relatively small. The manager does not intervene with Chord logic, and our system automatically transfers from manager mode to Chord mode to ensure the system efficiency does not drop. We also illustrate that the combination of the Chord system and an manager can produce useful applications upon Chord Protocol by building a file system upon the Chord protocol.**

***Index Terms*—P2P, distributed system, consistent hashing, replication, fault-tolerant**

I. Introduction

In this paper, we implemented a Chord-based key-value store distributed system based on the original paper. In addition, on the basis of the original Chord model, we added the data replication mechanism, where replicas are stored on each successor presented on the successor list of each node. The successor list is now used to maintain both the Chord node ring's functionality and the number of replicas in the cluster. Besides, inspired by the FAWN system, we implemented another centralized manager for the cluster. We then evaluated our system from correctness and performance, including the verification of the basic functionality of the Chord as well as the availability, efficiency, and fault tolerance.

II. Environment

A. Runtime Environment

Our System is running on 8 AWS m5.large instances. Characteristics of our configuration are:

TABLE I
Environment Configuration

Parameters	***Value***
OS Kernel	Red Hat 7.3.1
vCPUs	2
Memory(GiB)	8
Instance Storage(GB)	EBS Only
Network Performance	Up to 10 Gibits

B. Language: Java

The system is implemented in Java, which is fundamentally object-oriented, accelerating our development and making our code more intelligible and maintainable. Moreover, the very active Java community could provide us with more a robust and flexible solution in some aspects.

III. Methodology

A. Chord

Chord is a protocol and algorithm for a peer-to-peer distributed hash table [1]. In this system, nodes and keys would be assigned an *m*-bit identifier by using consistent hashing, which is based on SHA-1 hash function. Consistent hashing tends to balance the load, since each node receives roughly the same number of keys, and requires relatively little movement of keys when nodes join and leave the system. Upon obtaining the assigned identifier, a key would be stored The identifiers form a Chord ring like Figure 1 shows. Each Chord node contains a *m*-entry table called *finger table* in order to route across the network, helping maintain the time complexity of lookup operation in Chord as *O(logn)*.

B. Data Replication

In this section, we will discuss our data replication strategy. We start from our assumed failure scenario, then we talk about our choice of replication methodology. In section IV we will describe the implementation of our data replication mechanism. To simplify the illustration, we will use the word *primary node* to indicate the node whose node ID is exactly the result of consistent hashing the key of the data and we use *replica node* to indicate the node where data is replicated. We use the *primary space* of a node to indicate the data structure used to store its own data. The phrase *replica space* indicates the data structure to store replica data for other nodes.

Our data replication strategy is based on several assumptions. First, nodes do not persist the key-value data. Although data persistency is an effective solution, it cannot ensure data availability all the time. During the rebooting period of a failure node, the data is inaccessible to clients. Therefore, we argue we should replicate data on different nodes to ensure data availability. Second, we allow r-1 consecutive nodes in the ring to simultaneously fail, where r-1 is the length of the successor list. Meanwhile, the surviving node (which is the first alive node in the successor list of a primary node in our data replication strategy) will further replicate the data of the failed node to other nodes so that the replication numbers of all data are always guaranteed to be the user specified

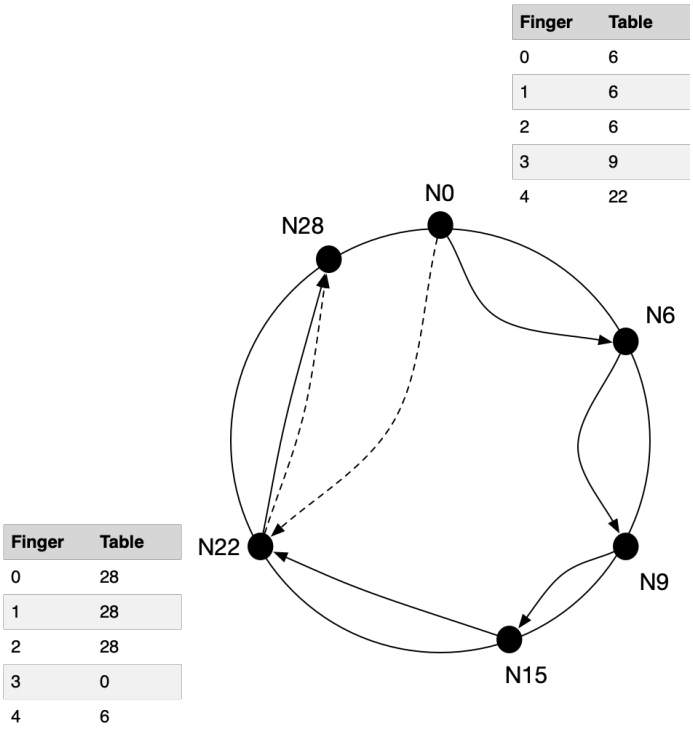

Fig. 1. A typical Chord ring

number. These replica nodes must be able to transform to the primary storage of data so that clients can always access the data. The third assumption we made is replica data cannot be served to clients. It indicates that we must separate the storage of replica and primary data. In our implementation, we store replica data and primary data in different HashMap. This policy may seem to be a waste of resources since it is a common strategy for allowing replica servers to serve data to clients is a good way to achieve load balance. However, consider the logic of Chord protocol. The Chord protocol first finds the successor of a requested ID. This successor is the only one in the system that is allowed to return the value of the key to clients. If the client gets the desired value from a replica node, it means the *findSuccessor* function goes wrong. Another reason for separating replica data and primary data is to help manage data replicas. Because we allow r-1 nodes to fail without affecting availability, it requires nodes to transmit data among the cluster. Sometimes a node may hold a replica of data that it should not have. For instance, node 7 holds the replica of node 3 in a system with the successor list length of 3, for node 3, 4, 5, 7 are consecutive in the Chord ring at this moment. Then node 6 joins the ring and node 7 no longer needs to hold the replica of node 3. Node 3 may contact node 7 to remove its replica. If all replica and primary data are stored in the same HashMap, node 3 must send a key set of its primary space to node 7 so that node 7 can remove the replica data. However, if the replica data is stored separately from the primary data, it can have a replica tag indicating the primary node of the replica. In that case, node 3 only needs to send node 7 its node ID. Node 7 simply remove the replica with tag 3 from the replica space.

Original Chord paper did not mention how to replicate data in the cluster. We argue it is indispensable to replicate data since the failure of nodes in an unreplicated system cause the data loss. We studied and considered three data replication strategies.

The first one is the most common strategy in distributed system world. It is simply copying the data of a primary node to several replica nodes. When the primary node fails, one of the replica nodes can be promoted to serve as the primary node. Because all replicas share the same data without any inconsistency, the replicas can also serve data with a load-balancer allocating requests for them. However, this strategy has an obvious defect. It requires n times of redundant machines to support this strategy, where n is the total number of replicas of data. In the real world, it could be a huge cost. To maintain the replica number of data, you simply reboot the failed machine or add a new machine to the replica group.

A simple improvement to the first strategy is to let a physical machine to play the role of both a primary node and several replica nodes of other primary nodes. This strategy is introduced in [6] as the overlay system K-Chord. K-chord maintains K Chord rings on n physical nodes. The i-th ring is the i-th random permutation of the same set of physical nodes. In other words, each physical node has different node IDs in these rings. The data object will be hashed to generate a node ID. This node ID is managed by different physical nodes in the K Chord rings. Then the data object will be stored to those nodes. As long as the hash function does not map two IDs to a single physical node, K-Chord can guarantee data to be replicated K times in the whole system. In the K chord system, the data only loses when all K replica nodes fail. To maintain the number of replicas, the successor of the failed node needs to find the data in other rings. It first looks up the key of lost data or replica in other rings, then copies those data from the storing machine to its local storage. During this recovery process, the data on the failed node is still available in other rings. So the client will try to search for the data in different rings until it gets the data.

The final strategy we take is the successor list replication strategy [3]. Successor list is only used to maintain the connection of the Chord ring in the original Chord protocol. To support data replication, a primary node's successor list also holds its replica. Because nodes will dynamically join and leave the Chord ring, the successor list of a node may keep changing until the Chord ring becomes stable. Whenever the successor list changes the replication of data will also change. To ensure the data of a primary node is replicated exactly on its successor list, we have several routine tasks conducted by all nodes.

measureDistance

Input: $node, sender, distance$
Output: $distance$
1: **if** $node.id = self.id$ **then**

2: **return** $distance$
3: **end if**
4: **if** inRange($node.id, self.id, sender.id$) **then**
5: **return** MAX_VALUE
6: **end if**
7: **if** $distance > sucLen$ **then**
8: **return** MAX_VALUE
9: **end if**
10: **if** not $node.successor.ping()$ **then**
11: **return** -1
12: **end if**
13: successor := node.successor()
14: **return** successor.measureDistance($node, sender, distance + 1$)

When the primary node fails, its successor will transfer the replica data from replica space to primary space. Thus the data can be served to clients. Note that we allow r-1 simultaneous consecutive node failure, which means, in the worst case, there may be only one node in the successor list survives and it could be the last one in the successor list. This node has replicas for all of the failed nodes as illustrated in Figure 2. So it can move all of its replicas to the primary space to take over all of its predecessors. Apart from data availability, we also require the redundancy brought by data replication to be as small as possible. It requires some other mechanisms. We will discuss these implementation details of our system in section IV.

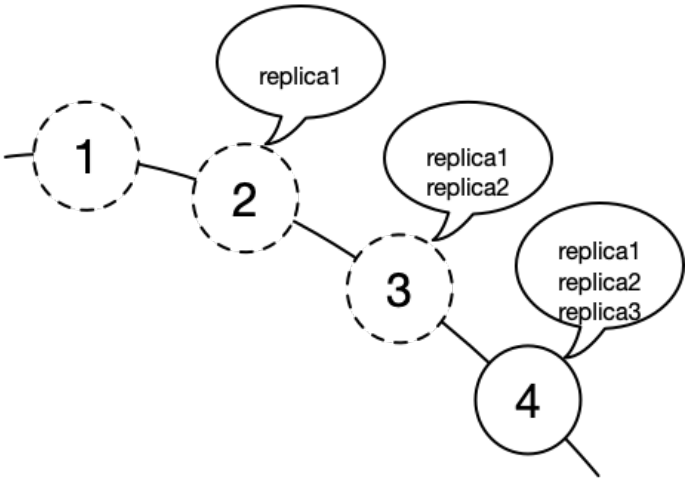

Fig. 2. Our replication mechanism. In this case, node 2, 3, 4 respectively has a replica for its predecessors. The dotted circle means these nodes fail simultaneously at this moment.

maintainReplica

Input: $newSucList, oldSucList$
1: $addedSucSet$:= $newSucList \cap oldSucList$
2: **for** suc in $newAddSucSet$ **do**
3: suc.addReplica()
4: **end for**

inspectRedundency

Input: replica
1: **for** $nodeId$ in $replica.keySet$ **do**
2: $distance$:= measureDistance($nodeId$)
3: **if** $distance$ = MAX_VALUE **then**
4: $replica$.remove($nodeId$)
5: **end if**
6: **end for**

C. Manager Node

A traditional way to implement a P2P system is to use a centralized manager to look up requested keys. Although it is simple, it is a single point of failure and requires O(n) memory to store nodes state information. Therefore, it is replaced by the Chord Protocol. However, if the cluster is of small size (e.g. 64 nodes in all), a centralized manager can largely improve the system efficiency. It only requires O(1) to search for the node to store a certain key(It degrades to O(n) if node number is far less than the ID range.). Moreover, the manager searches for node ID by traversing its state table, while the Chord Protocol searches for node ID by making a series of RPC calls. Thus, we add a manager to the Chord cluster as an alternative way to process requests. Note that the manager does not intervene with the Chord logic. And to avoid bringing the single point of failure issue to our system, our client automatically switches from manager mode to Chord mode. In other words, the manager only improves the system in a certain scenario but never cause the system to deteriorate.

Furthermore, we argue that a manager can be a useful interface for applications built upon our Chord Protocol implementation. Original Chord Protocol requires the key to be the digest of the file content. It makes the pure Chord system inconvenient for users to directly use, since a user has to know the content digest of a file to get the file. Therefore, we built a file system upon our Chord Protocol. The manager is used as a metadata manager of the file system. It manages a match from file names to the keys in Chord system. It also supports directory, so users can their files in the directory hierarchy. The filesystem makes file sharing and management on Chord Protocol easy.

IV. Implementation

Basic Chord implementation could be found in the original paper [1]. In this section, we mainly discuss our data replication mechanism.

A. Data Replication

As mentioned in section III, we adopted the successor list replication strategy. The two guidelines for our implementation are as follows:

- ***Guideline 1*** Suppose r-1 is the length of successor lists. All data should have exactly r copies in the system, no matter how nodes join and leave.
- ***Guideline 2*** The system must be able to serve the data as long as no more than r-1 arbitrary nodes fail.

Both guidelines are the highest requirements. The number of copies must be greater or equal than r to tolerate r-1 failures. Thus the first guideline means there is no redundancy in the

system. As for Guideline 2, the r-1 arbitrary node failures can be simultaneous failures happen upon r-1 consecutive nodes in the worst case. It means there is no time for the system to handle one failure before the next happens.

The basic replication implementation of our system has two routines. The first is that when a primary node's successor list changes, it will send the data in its primary space to all successors. The second is that a primary node propagates newly received data, no matter put by clients or inherited from the failed predecessor, to all its successors. Note the two routines only add replicas to the whole system. If we only have these routines, replication redundancy will occur when nodes join and leave as Figure 3 shows.

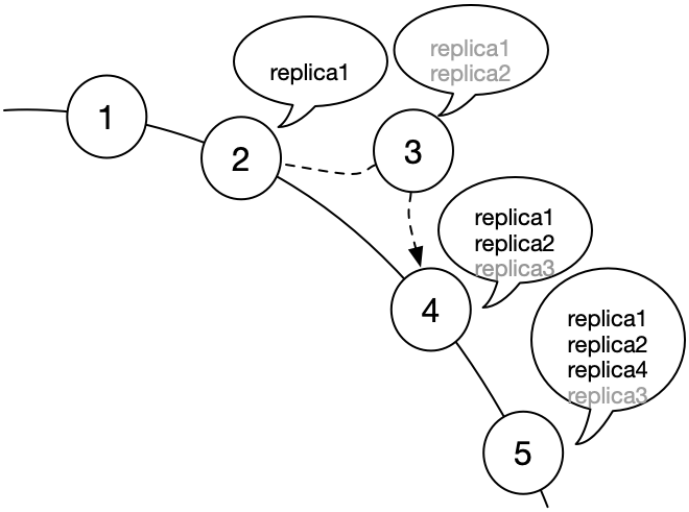

Fig. 3. A new node 3 joins the ring, so node 5 should remove the replica of node 1.

As shown in the figure, node 5 should remove replica 1. Therefore, we also have other routines that delete the replica from the whole system. When the successor list of a primary node changes, the node also tells those stale successors to remove its replica. Another routine is a periodic garbage collection method *inspectRedundancy*. It measures the distance of the replica node to the related primary node. If the distance is greater than the length of the successor list, the replica node will remove that replica.

B. Data Organization

The primary data is stored in a HashMap. This HashMap is the only data structure from which clients can access the data. Replicas are stored in a HashMap of HashMap as shown in Figure 4. The key of outer HashMap is the replica ID (which primary node it belongs to), the inner HashMap stores data. This structure helps us control replication redundancy, ensure data availability and reduce communication overhead in the following cases:

- *Case 1* When a node wants to conduct garbage data collection, it only needs to find the successor of the replica ID instead of all the keys in that replica shard.
- *Case 2* When some nodes fail, the replica ID can help the surviving successor to move related replica data to its primary space.
- *Case 3* When the primary informs a stale replica node of removing the replica, it only sends its node ID instead of all the keys.

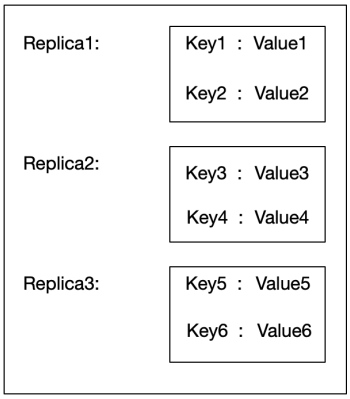

Fig. 4. Replica Data Structure

C. Availability

We require the data to be available when at most r-1 simultaneous consecutive node failures. Figure 2 is a case in which the replica amount is 3 (3 replicas and 1 primary). At this moment, node 1(primary), 2 and 3 fail. Node 4 must detect all of these failures so that it knows it should move replica 1, 2, 3 to its primary space. In the original Chord paper, nodes only check the aliveness of their direct predecessors. This is not enough for our goal, because node 4 may only move replica 3 to its primary space regardless of replica 1 and 2. Therefore, we rely on the *notify* method. When a node is notified that its predecessor changes to a node with ID i, it will remove all replicas with an ID between the new predecessor and the node itself. Then this node completely inherits and continues serving data on all failed nodes.

D. Optimization

For *findSuccessor* method, we implemented two versions: the iterative and the recursive. The difference between these two versions is the node who is responsible to forward the request, like Figure 5 shows. In the recursive version, the method is called node by node like a chain. And in the iterative version, every node return the query result immediately and attach an *isCompleted* flag bit to indicate whether it is the final result. Theoretically, the latter will give a better performance, since extra burden might be brought to the nodes in a chain in the former solution.

E. Manager Node

In this section, we will introduce the manager node we designed to boost the system in terms of the efficiency. By utilizing Java's inheritance feature, we extend a *chordManagerService* class from the original *chordNodeService* class, adding

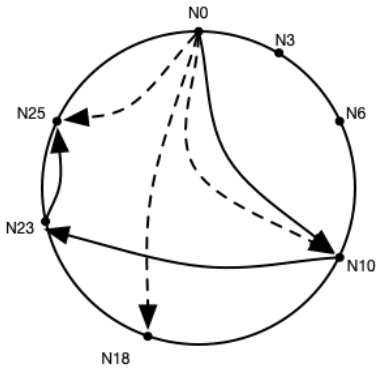

Fig. 5. Recursive Edition and Iterative Version of *findSuccessor*

extra RPC calls for nodes to contact with the manager in order to get the successor information. The *chordManagerServer* is the class for the manager node, which maintains a list of *NodeStatus*. We encapsulate each identifier information as well as its current status: online/offline within the *NodeStatus* class. By setting the field *status* to true, it implies that the current identifier has already been occupied and assigned to a Chord node. Upon Chord node initialization, instead of the *findSuccessor* RPC call, the node would ask the manager for its successor's information and the manager would first mark the *NodeStatus* of the input node as true, and then returns its successor by searching the list. Once it finds an active node, it returns its identifier; Else, it returns the node itself. The manager also sends heartbeat regularly to all identifiers to maintain its list information. The *PUT* and *GET* operation are also handled via the manager. The manager would decide a proper node to put or retrieve the data. In this way, the amount of RPC calls would be significantly reduced, though at the stake of a single point failure. We handled the single point failure by switching to the Chord service if detecting the failure, i.e, RPC timeout, and the original *findSuccessor* would be reactivated. During our experiment, we found that finding the successor via the manager is roughly 10 times faster than the Chord's *findSuccessor*, thus our manager could boost our system when the node number is small and has a low concurrency. When the node number is high, the manager might crash, and the base service would automatically take over.

V. Evaluation

The system is evaluated from two aspects: correctness and performance. For the implementation of test cases, we utilized JAVA JUnit testing framework to accelerate the process.

A. Correctness

In this section, we tested the correctness of our system under the circumstance of normal operation as well as the existence of failed nodes.

1) Normal Test:

- Test_Successor In this test, the correctness of the *findSuccessor* RPC call is verified. We first generate a local ground truth list with node's correct information, and then pick a random ID to call *findSuccessor* and verify its successor's correctness, iterating for 100 times. In this way, our implementation of *findSuccessor* could be determined.

- Test_FingerTable In this test, the correctness of our design of the finger table is verified. We first generate a local ground truth list with each node's information, and then iterate all nodes by calling *tellMeFingerTable* to acquire each node's finger table information. By comparing the ground truth with acquired real-time finger table, the correctness of our implementation of the finger table function could be determined.

- Test_Replica

2) Failure Test:

- Test_Successor_with_Failed_Node In this test, we focus on the correctness of our *findSuccessor* and the replica mechanism under the circumstance of failed nodes. For each iteration, a node ID is randomly picked and put a key-value pair. Then we manually kill this node and its r - 1 successors presented in the successor list, and verify if the user could still get the data from the last successor in the successor list. In this way, the correctness of our successor list and replication mechanism could be verified.

B. Performance

1) Redundancy Test: Our replica mechanism determines that a measure of redundancy will be brought into system under specific circumstances. And our test, based on node joining and leaving periodically, will give us a idea about the extend of redundancy. The test runs on a Chord Cluster whose node size is 256, and ring size 8192. We join and leave batch nodes whose size is 20. And the time gap is 20s. After the cluster stabilize method completes, we test the ratio of primary data and replica data. The result shows that there is no primary data loss, and the ratio is still 4.0(Our replication size is 3). It proves that our system could handle most of events happen in a normal Chord cluster.

2) Scalability Test:

- ***Latency***
  In this test, we evaluated our system's scalability by doing 500 times of *PUT* operations on different settings: 64, 128, 256, 512, 1,024 servers distributed across 8 AWS servers and then measure the corresponding latency. We compared the latency of the version with the replication mechanism and the one without replication, as well as the ideal scalable result. As shown in Figure 6, the system performs better than the baseline shown in red when the server number is 64, 128, and 256, i.e. at the low level

of concurrency. However, the latency surges when the node number increases to 512, especially at 1,024 servers, where the performance would be dramatically degraded. It is also obvious that the unreplicated version is slightly faster than the replicated one due to less communication overhead. The reason why the performance is degraded is the limitation of the machine resources. Due to the inherent Java Virtual Machine (JVM), if we opened 128 processes on each instance, it would take way more resources than 64 processes. Therefore, if we changed the AWS machine to a better one, we would have a much more scalable result when the node number is 1,024.

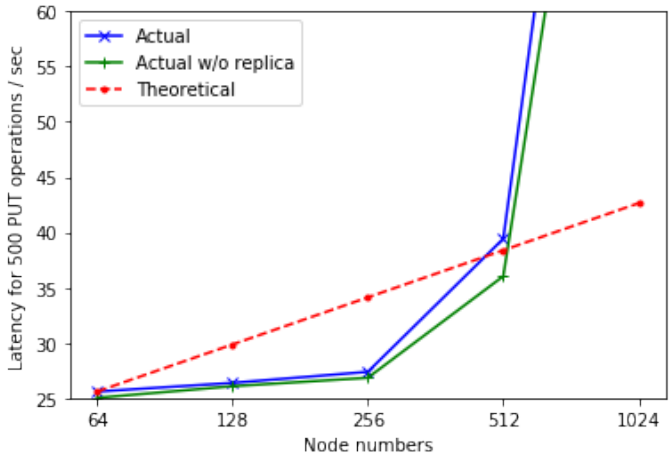

Fig. 6. latency of 500 PUT operations on 64, 128, 256, 512, 1024 node servers, with and without replicas

- ***Throughput***
  We also evaluated the scalability of throughput on our system. Our test script asynchronously issue 5,000 requests to a single node in the Chord ring. The cluster server number grows from 64 to 1024. The result is shown in Figure 7. The figures shows the throughput of 1024-node cluster, compared to 64-node cluster decreases by 40 ops/second. This is caused by the higher computation pressure on physical machines.

3) Path Length/Hops: In this test, we evaluated the average hops that our *findSuccessor* takes when searching for the successor. We first use the current timestamp to generate the node identifier, and find the successor corresponding to this node. By iterating 10,000 times, we could generate a hop frequency map, and measure the probability of the occurrence of each value. The node number is 512 nodes distributed across 8 AWS instances, with the ring size as 8,196. Table 2 illustrates the exact frequency that each hop occurs. Figure 8 shows the probability map of the hop number, which corresponds to the Chord paper's result.

4) Key Distribution: In this test, we measure the key distribution on nodes in our system by simply putting 10,000 random keys generated by the timestamp, and count how many keys each node have. We then plot the probability density

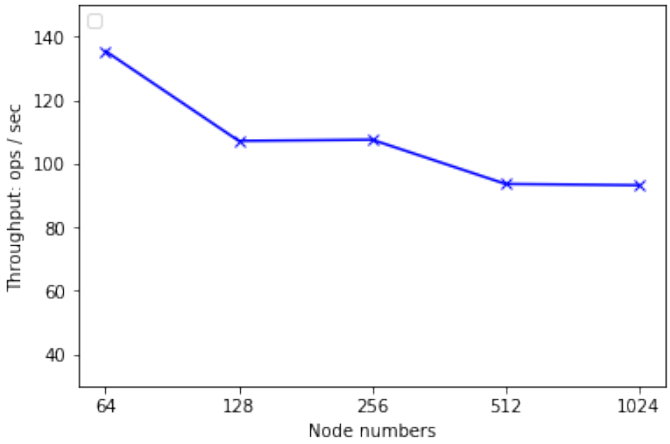

Fig. 7. Throughput on 64, 128, 256, 512, 1024 node servers

TABLE II
HOPS FREQUENCY

Hops	***Frequency***
1	18
2	232
3	953
4	2129
5	2537
6	2259
7	1272
8	505
9	95

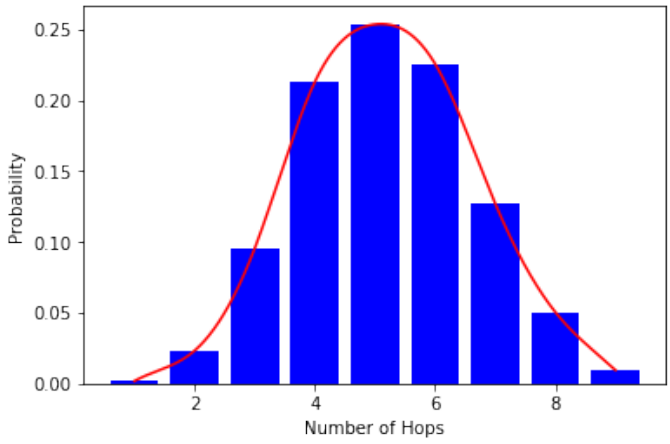

Fig. 8. hops probability for *findSuccessor*

function based on the number of keys per nodes, as shown in Figure 9.

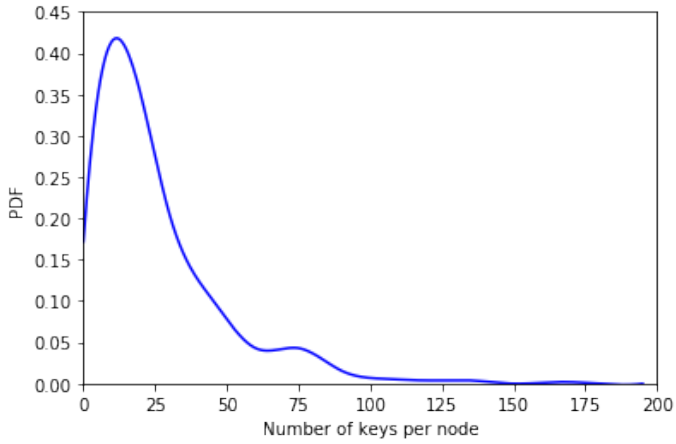

Fig. 9. the probability density function (PDF) of the number of keys per node. The total number of keys is 10,000. Total node number is 512, with ring size as 8,196

5) Comparison between Recursive and Iterative: Figure 10 shows the cost of time per 100 queries under different number of nodes. We could see that, it is obvious that iterative method performs better than the recursive one, especially for the large nodes number. When the number of node is small, the performance is similar but the difference becomes greater when the node number exceeds 512 nodes, which is because the longer query path means larger extra burden to servers. In the iterative mode, however, the burden is put on clients. Such result indicates the iterative *findSuccessor* solution clearly relieves the workload on server cluster. The result is as Figure 10 shows.

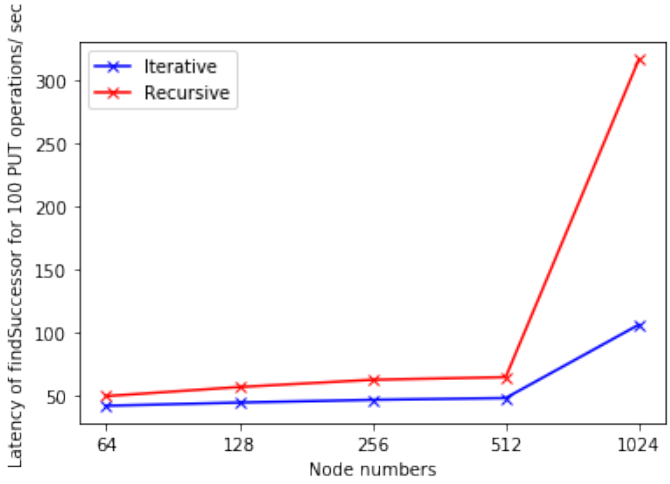

Fig. 10. Latency of recursive iterative calls

VI. Application

We built a file system upon the Chord Protocol making use of the manager. The overall architecutre is shown in Figure 11. The manager is used to manage the metadata of the file system. It maintains a directory tree. Each node in the tree is a directory which contains sub directories and "file descriptors" under current directory. The "file descriptor" here is the digest of the file content, or in other words, the key stored in the Chord system. Our file system client supports the following operations *ls*, *read*, *write*:

- *ls* When the user types in the *ls* command and an absolute path, the manager returns the sub-directories and files in the specified directory.
- *read* When the user types in the *read* command with an absolute file path, the manager returns the key of the specified file and the clients will automatically get the file content from the Chord system.
- *write* When user types in the *wrfite* command with an absolute file path and a local file name, the client stores the file content directly to the Chord system and contacts the manager to store the metadata of the file.

As a future work, we will add other commonly used features of file system. Such as change directory command, access control, user group.

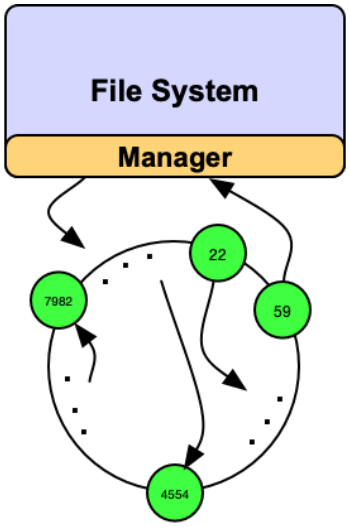

Fig. 11. Latency of recursive iterative calls

VII. Conclusion

In this paper, we implemented a Chord-based distributed key-value store system with data replication and manager node. The system achieves scalability, data replication and fault tolerance. The basic functionality of the system is like the Chord peer-to-peer protocol, which could evenly distribute nodes and assign them with identifier to a Chord ring by doing consistent hashing. We extended the system by implementing a manager node that maintains a list of identifiers on the Chord ring, accelerating the speed of *findSuccessor* as well as the *PUT* and *GET* operations. To fix the inherent single point of failure problem that the manager exposes, we let the service transit to the base Chord service logic once the node detects the failure of the manager node, although we found that

our manager node would only crash when the asynchronous request number is above 500,000 in 40 seconds. through our pressure test. We then evaluated our system in terms of its correctness, redundancy, performance and scalability through different test cases. On top of the combination of Chord Protocol and the manager, we built a file system to show the potential and possible usage of our system.

However, our implementation could be improved from many aspects. For example, during our research, we found some valuable ideas like chain replication [4] [5] and bi-directional [2] finger table. It is believed that such design could optimize the reliability and query efficiency, though some relative tests have to be done to verify this assumption. In the near future, we expect to do more explorations on these mechanisms and to see the difference that they might bring to our system.

REFERENCES

[1] Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.

[2] Vatsavai, Vasanthi, et al. Implementation of P2P File Sharing Using Bi-Directional Chord Protocol Algorithm. Lecture Notes in Electrical Engineering Advanced Computer and Communication Engineering Technology, 2015, pp. 5162.

[3] Dabek, F., Kaashoek, M., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storgae with cfs. In: 18th ACM Symposium on Operating Systems Principles (SOSP), New York, USA, pp. 202215 (2001)

[4] Flocchini, P., Nayak, A., Xie, M.: Hybrid-chord: A peer-to-peer system based on chord. In: Ghosh, R.K., Mohanty, H. (eds.) ICDCIT 2004. LNCS, vol. 3347, pp. 194203. Springer, Heidelberg (2004)

[5] Flocchini,P.,Nayak,A.,Xie,M.:Enhancing peer-to-peer systems through redundancy.IEEE Journal on Selected Areas in Communications 1(25), 1524 (2007)

[6] Kapelko, Rafa. "Towards fault-tolerant chord p2p system: analysis of some replication strategies." Asia-Pacific Web Conference. Springer, Berlin, Heidelberg, 2013.