A Peer-to-peer File-sharing System Based on Chord

***Abstract*—Efficiency, scalability and load balance are three fundamental problems when designing file sharing applications. This paper presents a BitTorrent-like peer-to-peer file-sharing system based on Distributed Hash Table (DHT) implemented using Chord. The system has a three-layer abstraction of its architecture: the low-level Chord infrastructure, the mid-level DHT with replication, and the high-level P2P file sharing. Chord adapts efficiently under variant internet conditions and can ensure load balance and scalability. Based on Chord, a weakly-consistent DHT with replication on successor-list is built providing Put and Get APIs. The asynchronous lazy replication strategy guarantees the eventual consistency of the whole system. Viewing it as a complete application, the BitTorrent-like design makes use of multiple sources over all copies of the content when downloading a file. Each active user contributes to the distribution of the content through their upload bandwidth so that the more users active on a torrent provides greater speed and reliability for each of the users. Experiments are conducted on Chord and Put/Get DHT APIs. We demonstrated the guarantees of the original Chord paper on load balance and lookup efficiency. We showed that the DHT is serving Put and Get requests with comparable response time, and the response time grows very slowly (sub-linear) when increasing the number of Chord nodes.**

***Index Terms*—Chord, DHT, chain replication, BitTorrent, peer-to-peer, file sharing**

I. Introduction

Peer-to-peer(P2P) file sharing is the sharing of distributed files, especially music, videos, games and other digits medias over peer-to-peer networks[1]. Different from traditional file uploading and downloading system based on the design with a clear separation of servers and clients, nodes in the P2P network can serve as both servers and clients. In a general way, every single peer node holds files in the local disk to share with other requesting nodes and can also locate other peer computers with target files and download from these peers with the support of a peer-to-peer file sharing program in each node.

In this paper, we designed and implemented a P2P file sharing system with a three-layer architecture based on Chord protocol. The first layer is defined as the routing layer which takes advantages of consistent hashing in Chord and efficient looking up with at most *log(N)* messages in an N-node network [2]. Above the routing layer is the DHT layer. It provides a distributed hash table service for storing and retrieving peer nodes addresses associated with a specific file block. As a hash table, the DHT also only exposes a Put and Get API while keeping all internal distributed states and optimization transparent to clients. Furthermore, a lazy replication strategy is proposed to achieve efficient chain replication in DHT. The third layer is the file application layer including the P2P file server and client program, a light-weight tracker which monitors the DHT cluster and provides to a client the address of an entrance node for the DHT service to a client.

The paper is organized as follows: Section II describes related work and background acknowledge about the P2P file sharing and Section III introduces the design of our system in a high level. The implementation and details about each layer are presented in Section IV, followed by the experiment and evaluation of the system in Section V. Finally we draw some perspectives about the future work and make a conclusion.

II. Background And Related Work

A. Chord

Distributed hash table (DHT) is a decentralized system that provided a lookup service similar to a hash table. Based on DHT as an infrastructure, many complicated applications can be built such as Domain Name Services (DNS), distributed file systems, etc. Peer-to-peer lookup protocols such as Chord [2], CAN [3], Pastry [3] and Tapestry [4] are often used to implement a DHT. The original Chord paper introduces how data are assigned to nodes and how requests are routed in Chord system, as well as the scenario which nodes might join and leave the system. Chord provides several good features: the consistent hashing guarantees that only minimal work is required to maintain the system with respect to system topology changes (nodes join and leave); it has good scalability—each node maintains information about only *O(logN)* other nodes, and a lookup requires *O(logN)* messages; the algorithm to spread data storage and request handling to all nodes provides the functionality of load balancing.

B. Replication

Building reliable DHT requires data replication to tolerate either network disconnections or server failures. We analyzed the strength and weakness of several popular replication strategies: successor-list, leaf-set, multiple hash functions and symmetric replication. The location of replicas for these four strategies are:

- Successor-list: replicate data to all nodes in successor-list, which is consisted by all f (replication degree) successive nodes including the primary. This strategy mentioned in Chord [2] and implemented in FAWN [4];
- Leaf-set: replicate data to leaf-set, which is a set of nodes logically centered around the root node. The maintenance effort is the same as successor-list. This is implemented in PAST [5];
- Multiple hash functions: use multiple hash functions to determine a set of nodes to replicate data. This is used in CAN [3] and Tapestry [6];
- Symmetric replication: using symmetric replication, which is partition all nodes in to several groups and every node in each group acts as a replica for other nodes. Each identifier is bidirectional associated with f other

identifiers. A carefully designed partition should allow a node to identify its associates by a simple calculation. This is proposed and implemented in DSK [7] and is also adopted in Scalaris [8].

Two factors determine how efficient the strategies are when performing recovery under node join and leave: efforts needed to identify which is the destination of data transfer and the amount of data to transfer. For all four strategies, when a node joins or leaves the system, only entries that reside in one single node will be transferred or erased to maintain the replication degree. For successor-list and leaf-set, since the replicas all have successive logical addresses (identifiers), it is very easy for nodes in successor-list and leaf-set to have a clear view of the whole set, so that when a node detects a failure of neighboring nodes (which is already there in Chord to update successor and predecessor), it can just create a new replica to the end the whole set. For symmetric replication, although nodes in the same partition can identify its associates by performing some computation, since nodes in a group are not necessarily successive in addresses, it requires additional heartbeats to monitor the liveness of associates. The advantage of symmetric replication is that it allows to make use of physical distance of nodes to preparatorily improve the locality of replica groups by carefully designing the partition. Its obvious that locality of replicas improves the efficiency of replication. While as for successor-list and leaf-set which use successive logical addresses, it is at the discretion of hash function to almost-randomly assign logical addresses, which might assign successive addresses to two nodes that are distant from each other. For multiple hash functions strategy, nodes cannot detect failures of other replicas either and need to have additional heartbeats to do so. Additionally, knowing the addresses of other replicas that host the same entry is difficult. Considering that key=X of a key-value pair is hashed to node A and B using two different hash functions f1 and f2. A must have a sense of the liveness of B in order to start a new replica storing this key-value pair in case of B fails. But A cannot hash back using f1 to get key X since hash function is unidirectional. In summary, we see that using successor-list is intuitive and efficient, so we choose to use successor-list as our replication strategy.

C. P2P file sharing

Among many applications that take DHT as infrastructure, we are particularly interested in the P2P file sharing system. The Napster is a famous P2P music sharing system which has a central server to store metadata of files including the filename and addresses of peers that host the file. Clients (peers) can upload files by telling the server the filename and its network address. Other clients can lookup filename on the server and get the address of nodes hosting that file and contact those nodes to download the files. It's a typical P2P system since a client can be downloading file A and uploading file B to some other clients. This simple design suffers from single point of failure. A decentralized P2P file sharing system Gnutelle does not rely on a single server. It is based on flooding the lookup requests to the whole network to find the nodes which has the specified file. Nodes broadcast requests to all its neighbors and eventually a node in the network replies. Obviously, this approach of lookup is inefficient and consumes lots of bandwidth.

BitTorrent is possibly the most successful P2P file sharing system. By 2013 it is the largest P2P system with 15-27 million concurrent users online [9]. To download a file given only a filename, clients first need to get the 'torrent' file from a server usually through a website. The 'torrent' file contains metadata of the file, such as digest of each file block which can be assembled to obtain the complete file, and the address information of a tracker server. The client now knows which block to download, and it contacts the tracker to get the addresses of a list of nodes which actually host the blocks. Finally, the client knows the block to download and where to download. The lookup of addresses of a list of nodes (value) given a digest of block (key) is the heaviest traffic in the system, and also is the most dynamic system since peers come and leave very frequently—when a block has been downloaded, the client can become a server to host the block. This scenario perfectly matches the strength of DHT.

III. Design

The project is a distributed BitTorrent-like file-sharing system. The intention is to distribute large data and electronic files over the Internet. Combined with scalable peer-to-peer lookup protocol and fault tolerance technology, the whole project can be divided into 3 layers from infrastructure layer to application layer. But before diving into the specific details of each layer, we first introduce the architecture of our system using a high-level big picture.

As is shown in figure 1, there are 3 memberships in our system: tracker, chord nodes and peers. Tracker is a server with fixed address which is known to all other members in this environment. Chord nodes are responsible for information lookup with balance load, which can be deployed on AWS servers in different continents. The peer in this distributed system plays two roles: client and file holder. In order to download a file, the client needs to obtain entrance address to Chord ring and hashed value of target file from tracker. Considering the parallel downloads, we optimize this process further by breaking down a whole file into small pieces or *blocks*. Thus, the client should obtain the hashed value of a block of a large file from the tracker. After that, client contacts chord nodes and receives a address list in which there are all fileholders' addresses. And in the end, the client picks one of valid addresses in the list and grabs block data from a fileholder directly. By downloading all blocks from different fileholders, the client can merge these pieces up locally and generate a complete file. Similarly, for uploads, fileholders contacts tracker and lets it to register its address and holding blocks in the Chord ring.

Based on the above analysis, we decided to divide the whole system into 3 layers to implement. The first layer focuses on Chord. In order to realize high scalability, Chord

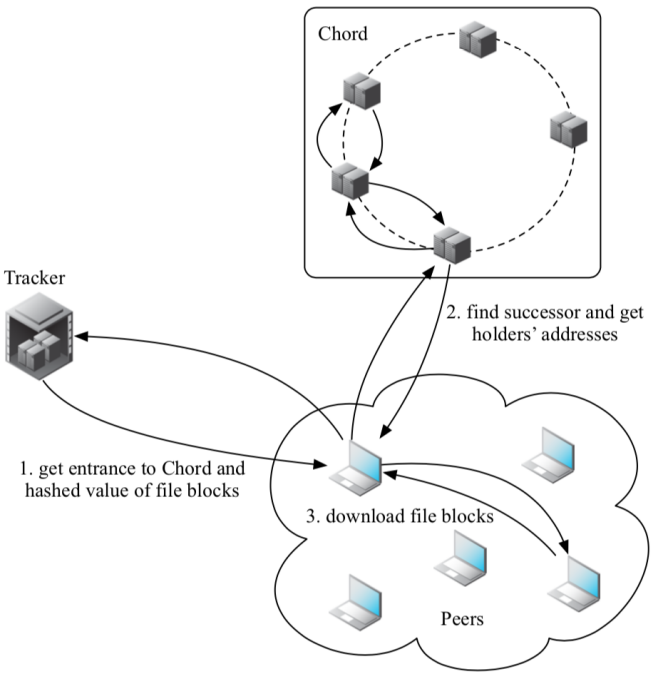

Fig. 1. High level picture of BitTorrent file sharing system.

protocol is chosen naturally due to its appealing attributes like load balance, decentralization, scalability and availability. We implemented all technical details including successor finding, finger table maintainance and node join/leave according to Chord paper[2].

Second layer is *distributed hash table(DHT)*. At first, we considered storing file data in each chord node. But this is high-cost and violates peer-to-peer principle. Thus, each node stores mapping of file block key to a list of fileholder addresses. Everytime a peer upload or download a file, its address and corresponding file block keys will be recorded in chord nodes. In order to ensure fault tolerance, we realized a replication chain mechanism according to [4]. But we optimized it based on original implementation to ensure asynchronous lazy replication because our main purpose is load balance instead of consistency. We set up a *checker* in each node which periodically sends heartbeats to its successor piggybacking replication information pairs. In order to indicate updated address list of a existent block key, we introduce a new variable *sequence number* which increments monotonically by 1 for each update. For example, if the replication chain length is 3 and one primary chain node stores a mapping of *hashed_key_of_block* to *address_list*, its checker will periodically sends the pair (*hashed_key_of_block*, 2, *current_sequence_number*) to the successor. The successor would fetch this updated or missing mapping if the received sequence number doesn't match or the successor doesn't store this necessary information. Besides, the second layer involves concurrently modifying shared data and needs a lock to ensure consistency. But we considered the special attributes of P2P application and found it was possible to remove the lock for read operations so as to increase throughput.

Main application functions are realized in the third layer. Since this is a P2P BitTorrent distributed system, we need a tracker to coordinate communication between chord nodes and peers. Obviously, the tracker is a bottleneck if the whole system scales up. Additionally, the flow rate on Chord ring should be larger so that an extra tracker storing metadata is a great choice. In this case, we design the structure of tracker as simple and light-weight as possible. The tracker maintains a list of active online chord nodes which is provided as entrance to chord ring. Every time a node joins or leaves, the predecessor would notify its new successor and send a update message to tracker to add in or remove from this anomaly address in the active chord node list. Furthermore, peers have functions like download, upload, put, get and debug.

After illustrating used mechanisms in the system, the whole workflow can be organized in the figure 2. There are collection of chord nodes and a collection of peers. When a client wants to download a file, it firstly get entrances address, e.g. x, from the tracker. Then the client sends a *find_successor* request to node x which subsequently looks up the finger table and invokes the same interface. In the end, the predecessor y of the target node z returns the real address. In the next phase, the client contacts node z directly and obtains a list of addresses which all have the target file blocks. The client picks one of them and downloads the block.

In the next section, we will introduce complete technical details for each layer involving data structure, application interfaces and in-depth implementation of functionality.

IV. Implementation

We followed the three-layer design to implement the file sharing system. In the infrastructure layer, we implemented a basic Chord system according to the original paper [1] with a few modifications which improve the efficiency. In the DHT layer, we adopted the successor-list data replication strategy and implemented an asynchronous lazy replication mechanism which guarantees eventual consistency. In the application layer, we implemented a tracker to serve file metadata just like BitTorrent and a P2P client program to meet high-level file upload and download requirements. Communication among client, tracker, and Chord nodes are all implemented using gRPC in Python.

A. Infrastructure layer

The Chord infrastructure consists of two main parts: the routing algorithm including finger table that expedite the routing, and the efforts to maintain the finger table with respect to network topology changes, including nodes joining the present Chord ring and leaving the system abruptly without notifying other nodes beforehand.

The routing procedure is implemented exactly as described in the original Chord paper. The hashing of keys (either nodes network address or the key in key-value pair to access the DHT) to identifiers is performed using SHA-1 followed by modulation of hyper-parameter M. The logical topology of a

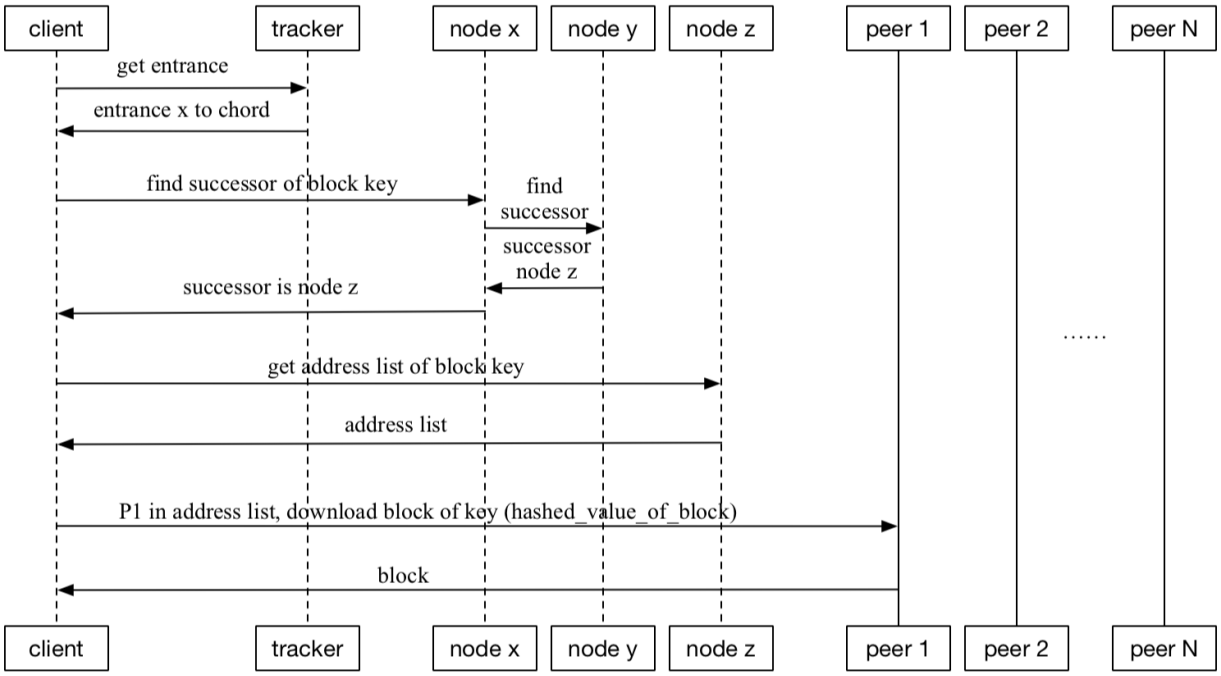

Fig. 2. Time sequence workflow of BitTorrent distributed system.

Chord ring is determined by this hashing process to construct an overlay network. There are at most 2^M nodes in Chord ring. The RPC function *find_successor(id)* returns the result if the target identifier is within the range of its successor and itself; otherwise the RPC issues a new RPC to another node according to *closest_preceding_node(id)* which searches local finger table. The RPC relay works recursively. The newly issued (inner) RPC within current (outer) PRC blocks the until the inner returns.

There are three import functions that guarantee the correctness of routing: *stabilize()*, *notify(id)* and *fix_fingers()*. *stabilize()* and *fix_fingers()* execute periodically in background, and notify(id) only execute when a node wants to change its successor and let the new successor know this change. We find that having the *check_predecessor()* as is described in Chord paper is not necessary since the liveness of predecessor can be indicated by the received notify() call. Specifically, every node calls *stabilize()* periodically to check successors liveness. The alive successor acknowledges with the information of its current predecessor piggybacked. Therefore, *stabilize()* also enables nodes to detect a newly joined node between itself and its successor. In the case where successor does not reply, a failure of successor is detected and it calls *notify()* to contact the second successor. This is like saying Hey! Your predecessor has failed. Now I am your new predecessor. *stabilize()* and *notify()* provide a node with the ability to monitor the liveness of its neighbors. To have a necessary global view of the topology, the correctness of finger table is periodically checked through *fix_fingers()*. This function uses *find_successor()* to check the correctness of every entry in finger table. As long as the correctness of local connections (predecessor and successor) is met, *find_successor()* will always get the correct result even if the finger table is incorrect.

With these three functions, the join and leave of nodes can be easily handled. A newly joined node only needs to find its successor by calling *find_successor()* and notify it. The periodical *fix_fingers()* will soon fill the finger table correctly. When a node leaves without any deliberate notifications, its predecessor by stabilize() will soon find that it is not responding and replace it with the second successor. The address of the second successor is already stored in the second entry of finger table.

B. DHT layer

Based on the Chord protocol, we build up a Distributed Hash Table(DHT) which can store and retrieve $< key, value >$ pairs over a network with distributed peer nodes while providing scalability and eventual consistency. Basically, the DHT only provides two primitives to the clients: *Put(key, value)*, *Get(key)* while all of the internal communication and functions, such as replication, are transparent to clients.

A client can contact any nodes in the cluster as the entrance to get ID and address of its successor node by invoking the *Find_Sucessor(key)* RPC as last section shows. Once the client obtains the matching successor address, it will directly call the *Put(key, value)*, *Get(key)* RPC to store or retrieve mapping $key, value >$ pairs. So the client only cares about the result and correctness of PUT/GET operations and has no idea what it is like in the internal DHT cluster.

Currently, for simplicity, the whole DHT storage is based on the memory without involving any disk storage. For each node,

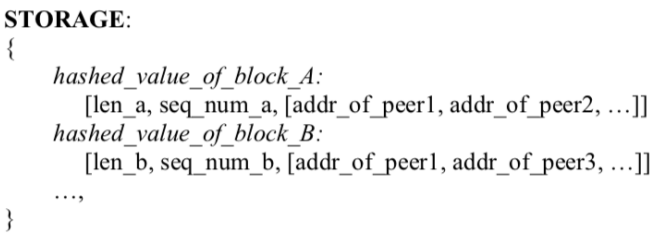

Fig. 3. Data Structure of Storage in each chord node

it maintains an in-memory dictionary, defined as STORAGE as figure X shows, to store all of pairs. Here, A key is a hashed value represented as a hexadecimal string by hashing the content of a file block using SHA1. For example, a block may be hashed to be ABC13F341 as the key. There are three items for each block *len*, *seq_num*, *address_list* where *len* and *seq_num* are related to the replication chain and *address_list* is a list of address about all of the peer client computers that currently hold this block in their disk. Notice that different from the traditional Hash Table where a PUT operation directly overwrites the old value, our DHT instead appends the client address to the *address_list* related to the key. This difference results from the fact that there are several different clients holding the block at the same time and the DHT should record all of the possible client nodes. Thus it takes the append instead of the overwrite operation.

We implemented an async lazy replication strategy that provides eventual consistency while the external consistency is not guaranteed in our system. The overall replication strategy is based on successor_list, which means saving multiple replicas in several successor nodes just after the target node that the data should be stored originally in DHT. We mark this target node as *Primary* while nodes in the successor_list are marked as *Back Up*. Each time when a *Primary* receives a request for putting a key-value pair, this *Primary* will firstly update the local Storage and then returns to client. At the same time, a *checker* thread will periodically send messages to ask its successor to keep in update with its newest state.

1) len: In our replication chain, each block in a node is associated with a *len* field indicating how many replicas left in the replication chain. For example if *len* is set to be 3, it means there is still 3 valid replicas starting from current node in the replication chain(including itself). In other words, there should be another two nodes holding the replica of this block after this node. This field is used to provide hints about where the replication chain should end and when the unnecessary replica over R should be deleted when a new node join in. Here, *R* is the replication degree. Suppose the replica degree is set to be 3 and now it has been stored in three nodes as B,C,D. When a new node A joins in just before B, some data in B should be remapped and moved to node A, thus the extra replica in node D is not useful anymore. When D receives a message from C indicating the *len* is 0 for this block in D, node D is now aware of the fact that there have been enough replicas in the nodes before it and thus delete the block from its local Storage.

2) seq_num: We designed another field *seq_num* for each block in the Storage to let the successor node know when it should fetch data and keep in update with its predecessor in the replication chain. Initially, the *seq_num* is set to be 0 for a new block that has just been put from client. Each time when a client contacts the *Primary* and updates the *address_list* related to a block a, the *Primary* should increase the *seq_num* by 1. Afterward, the checker process will send the block information with new *seq_num* to its successor. Once the successor receives this update, compares the *seq_num* and is aware of a larger *seq_num* in its predecessor, then it knows that the local replica about block a is out of date. Thus, it invokes a Get RPC to its predecessor, fetches all of data related to block a and updates its local Storage.

3) checker: Inspired by the *stabilizer* and *finger_fixer* in Chord paper, we use another *checker* thread in each node to periodically send messages about its current states to its successor for the asynchronous replication. Instead of sending all of data in Storage, the checker only extracts and sends the information about the *len* and *seq_num* of each block to increase efficiency and save time. When its successor receives the checker message, it will update its states by several steps. Firstly, if the received *len* for a block is 0, then it will directly delete the block from Storage. Otherwise, it updates the *len* field locally to be the given *len*. If the block has not been deleted, it compares the *seq_num*, fetches all out-of-date blocks from its predecessor and updates local Storage. For each checker process, the successor should keep its states updated with the corresponding states that its predecessor suggests. In this way, the replication is passed one by one until the end of the replication chain.

C. Application layer

The main topic of application layer is how to develop tracker and peers. As mentioned in design section, the core idea of tracker development is to make it as simple as possible. Thus, the tracker in this system only shoulders two missions. One mission is to maintain a *chord_nodes* data structure which represents the current active nodes in Chord ring. When a node joins in and leaves from the ring, *rpc_add_chord_node* and *rpc_remove_chord_node* would be invoked and corresponding variant node's address would be added or deleted. Furthermore, it also uses a hash table data structure to record file name and its corresponding hash value. File block registration and lookup can be realized by *rpc_register_file* and *rpc_look_up_file*. Another RPC call is *rpc_get_entrance* used to get a random address from active node list as an entrance to Chord.

The functionality of peers are mainly based on Chord with the help of tracker. A peer can play two different roles in two scenes. It can be a client when receiving file blocks and can also be a fileholder when transmitting file blocks. There are some local functions which is used to invoke remote RPC calls. At first, there are two basic functions *put* and *get* which

are used to store the mapping of file block hash value to current address into chord node and get this mapping from target chord node. Besides, client can invoke *download* to download a file block from remote peer and can also become a fileholder since then. And it can also invoke *upload* to register as a filehoder.

V. Experiment And Evaluation

A. Load Balance

Load balance is a great advantage of Chord DHT. Ideally every node in the Chord ring stores the same amount of key-value pairs, so that clients' requests are evenly distributed to all nodes in the ring. We conducted an experiment to see how the system actually works. We set up a 8-node Chord ring with M = 5, which means there can be at most 32 nodes and the every node is responsible for 4 different identifiers on average. We assign random port numbers to 8 nodes, making the distribution of nodes in logical address space relatively random. We also randomly generate 100 to 1000 key-value pairs to put on the Chord DHT. The randomness of node distribution and request schema are expected to simulate the complicated scenarios in real life applications. The replication degree is 3 in this experiment.

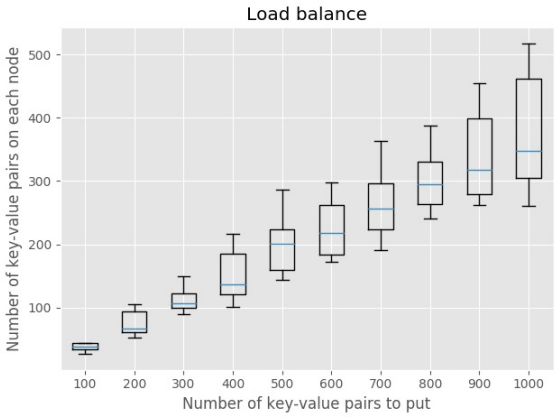

Fig. 4. Mean, 1st, 25th, 75th, and 99th percentiles of the number of key-values pairs per node on an 8-node Chord ring with M = 5

From figure 4, we observe that the maximum and minimum numbers of pairs per node are nearly in proportion with the total number of pairs, and the variance of number of data per node is relatively large. When putting 1000 key-value pairs on the DHT, the maximum number of pairs is around twice as large as the minimum number of pairs. The number of pairs at different percentiles has a trend of growing linearly. This damages the performance of load balance in large number of requests. Regardless of the pattern of clients' requests and assume they uniformly cover the address space, the distribution of node ID in the address space can be optimized using virtual nodes.

B. Path length with different nodes

The Chord protocol provides an efficient looking up for DHT with at most log(N) routing messages to resolve a *Find_successor()* RPC request in a N-node system. In this experiment, we changed the number of real nodes from 2 to 16 in a 32-Node network(M=5) and recorded the *path_length* in each looking up. For every cluster size, we experimented randomly with 1000 *Put* operations and 500 *Get* operation to explored how the *path_length* changed. The result is illustrated as Figure 5.

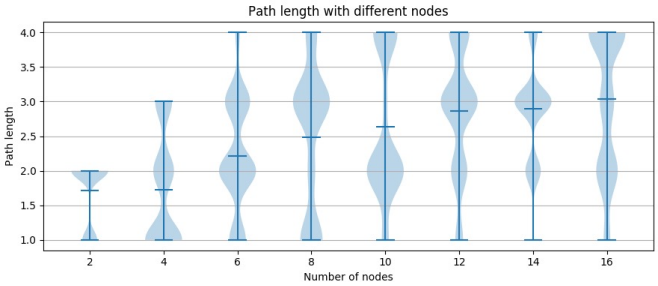

Fig. 5. Max, average, min of path length with different nodes on a Chord ring with M = 5

Figure 5 shows the max-min-average value of path length with different number of nodes. In this experiment, the min path length is all 1, which is reasonable. We have experimented 500 times with each configuration, so it is highly possible that we are looking up a key exactly stored in the entrance node, thus the path length is 1. The average path length slightly increases with the node number due to the fact that data is distributed to more nodes, thus more queries is needed to get the successor of a give key. Notice that the max path length changes from 2 to 4 and none of them has exceeded the max limit of 5(log(32)), which meets our theoretical analysis.

C. Replication complete time with different chain length

In this experiment, we measures the elapsed time needed to replicate file blocks on all replication nodes with different replication chain length. Since the replication in our system is realized in a lazy asynchronous way, we compared the time stamp between primary node and tail node in the replication chain. When M = 5, i.e. 32 node space, we experimented with chain length ranging from 2 to 6 and the result is shown below.

According to the figure 6, the length of replication chain goes up, the time needed also increases. Since the replication process is not parallel, the following node completes replication after the preceding nodes complete. In this case, the longer replication chain is, the longer time it takes. Thus, finding a trade-off between efficiency and satisfied fault tolerance is crucial.

D. Number of Nodes and response time

Since our system is not a pure load-balanced distributed system, we need to pursue short response to complete file uploads and downloads. The underlying layer relies on Chord protocol which uses consistent hashing and whose structure is a ring. Because of this, same hashed key may be stored in different nodes if the number of nodes in the system is

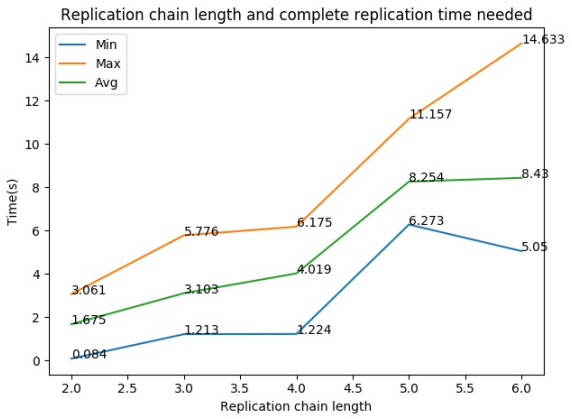

Fig. 6. Replication complete time with different chain length

variant. We changed the number of real nodes in the ring and tested average time of 1000 times of put and get requests on condition that the length of replication chain is 3.

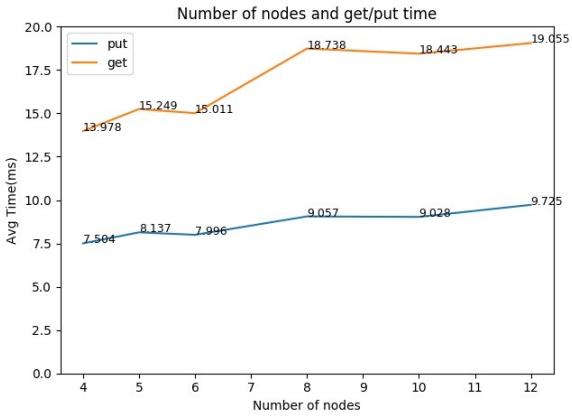

Fig. 7. Number of Nodes and response time.

According to the result, we can find that as the number of nodes increases, the system needs take longer time to response. Since the most time consuming operation in this process is to find the successor, if there are more nodes residing in the ring, the average path length of finding the successor will be longer and it may take more time. Another feature is get request takes more time than put request. That might be the result of return values and RPC language features. Since the purpose of get request is to get a list of fileholders' addresses and we used grpc to realize RPC interfaces, there might be extra time to transfer the address list from grpc message data stucture to local data structure. Thus, get requests are more time consuming.

VI. Future Work

We have designed a lazy asynchronous replication strategy to efficiently achieve the replication chain. A checker thread in each node will periodically send block meta-data to the successor node so that the successor nodes can keep updated with the primary eventually. But since it is asynchronous, if the primary crashed before its checker successfully sent the updated block metadata to its successor, then the modified states in this primary would be lost. We have not come up with a very good solution to this problem. A possible alternative is that a primary can send a replica to its successor and return OK response to the client only when it has received an acknowledge from its successor, indicating the successful replica. Afterwards, the checker thread will automatically pass the updated data information to the rest of nodes in the replication chain. In this case, our system can survive the failures when any neighboring nodes don't crash at the same short-period time.

Other optimization can be applied to the checker process in our replication design. When a node joins the Chord ring, its successor is responsible to transfer data entries to the new node. These entries by hash function belong to the new node. In our current system, every node has only one bucket for all identifiers it is responsible for. Therefore, the successor must look through all local entries, calculate their identifiers and check whether they belong to the new node. An optimization is to have different buckets for entries with each unique identifier. This avoids reading all local files and improves the availability of the system when Chord ring changes.

VII. Conclusion

Many distributed peer-to-peer applications need to find the node that stores information. The chord protocol can solve this problem and bring something more because of its appealing features like consistent hashing and flexibility. It offers one simple but powerful primitive: it can map a given key onto a chord node. In the steady state, the searching path length in chord ring is O(log(N)) order of magnitudes for an N-node network, which perfectly satisfies our requirement. By combining chain replication and consistency, our file-sharing system can provide high reliability even when a new node joins into the network or an original node leaves. Any failure or error in the chord ring can be corrected by either predecessor or successor in our mechanism.

Considering the application value of the system, we also implemented BitTorrent protocol on application layer. By breaking down a large file into small blocks, each small file contains information about specific (associated) content, which allows client to download a large file in parallel. In order to avoid the potential constraints brought by centralized tracker, we made it as simple and light-weight as possible so as to improve throughput and reduce latency.

Our theoretical analysis and simulation results confirm that this peer-to-peer file sharing system scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and ensures fault tolerance via chain replication. Admittedly, there are some future work to make it better. But we believe that our system is a valuable application for peer-to-peer large-scale distributed file-sharing system like cooperative file sharing and storage.

REFERENCES

1 Ripeanu, M., "Peer-to-peer architecture case study: Gnutella network." Proceedings first international conference on peer-to-peer computing. IEEE, 2001.

2 Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H., "Chord:a scalable peer-to-peer lookup service for internet applications." IEEE, 2003.

3 S. Ratnasamy, M. H. R. K. and Shenker, S., "A scalable content-addressable network," in *Proceedings of the ACM SIGCOMM Conference, 2001, pp. 161172*.

4 Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V., "Fawn: A fast array of wimpy nodes." ACM, 2009.

5 Rowstron, A. and Druschel, P., "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," in *ACM SIGOPS Operating Systems Review. Vol. 35. No. 5. ACM, 2001.*

6 Rowstron, A. and Drusche, P., "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), 2001, pp. 329350. [4] B. Y. Zhao, L. Huang, J. S.*

7 Ghodsi, L. O. A. and Haridi, S., "Symmetric replication for structured peer-to-peer systems," in *Databases, Information Systems, and Peer-to-Peer Computing. Springer, Berlin, Heidelberg, 2006. 74-85.*

8 Schtt, F. S. and Reinefeld, A., "Scalaris: reliable transactional p2p key/value store," in *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG. ACM, 2008.*

9 Wang, L. and Kangasharju, J., "Measuring large-scale distributed systems: case of bittorrent mainline dht," in *IEEE P2P 2013 Proceedings. IEEE, 2013.*