Chord Peer-to-Peer Lookup Protocol Implementation and Measurement

***Abstract*— In this research project, we implement Chord, a scalable peer-to-peer lookup protocol to provide KV-Store service using DHT(Distributed Hash Table). Chord Ring consists of 2^m possible node IDs, where server nodes map their server name (or IP) to a node ID. Both of our server and client applications are implemented with Python and gRPC. Additionally, a Chaos Monkey feature is integrated to simulate that nodes dynamically join/leave the Chord Ring. Our implementation supports nodes joining and leaving the system, either intentionally or due to failure, our Chord Ring implementation refrains from losing data with failure of k nodes. Our experiment and evaluation are done locally and on AWS EC2 with different number of servers. Evaluation covers load balancing, lookup path length, stabilization time with dynamic node join/exit, data replication response time. To prove the correctness of our implementation, various cases including edge cases were designed.**

I. INTRODUCTION

This report first talks about our implementation of Chord, focusing on how it differs from the original paper, discusses about possible future improvement, then evaluates performance and tries to give proper reasoning, at last we would prove its correctness.

The Chord protocol is based on consistent hashing, a key is assigned to the node whose identifier is equal to or first follows the key in the identifier space(the first successor of key). Each node stores a finger table of size m, a successor list of size r and a predecessor. We implement the create() function and the join() function. When a node is the first node in the Chord Ring, it will call create() to establish the Chord Ring. A node can join the Chord Ring given any known node in the ring. Our implementation of Chord server has stabilize/fix_finger/check_predecessor threads which are controlled by Condition Variable to ensure only one runs at a time. We create three daemon threads to periodically call these functions. The stabilize thread is responsible for verifying node's immediate successor and telling the successor about the node. The fix_finger thread refreshes finger table entries and the check_predecessor thread checks whether predecessor has failed. We implement the Chaos Monkey feature to make nodes dynamically join and leave the network.

Our implementation is proven to be correct, load balancing, scalable and available. We prove the correctness of our implementation theoretically and by designing a series of edge cases. Chord spreads keys evenly over the nodes and we introduce virtual nodes to improve load balancing. For scalability, the lookup path length and the size of data stored on each node grows as the log of the total number of nodes. Very large scale of Chord Ring is feasible without much overhead for system maintenance and key lookup. For availability, the node responsible for a key can always be found even if the system is in a continuous state of change. And we implement a successor list of size r to ensure that the system won't break down with less than r nodes failure.

We implement data replication on top of Chord protocol. To ensure there is no data loss with k nodes failure, we need 2k+1 replication of the data(including the original one). Once receiving a PUT request from client, we first locate the node responsible for that key. Then we copy the key to the node's successor list with size 2k+1. After receiving k response from nodes in the successor list, the node responds to the client.

During research, How to Make Chord Correct is referred to during implementation. We make some modifications and optimizations in our protocol from the original paper. We combine the original join and reconcile function into a single join function, combine original stabilize, reconcile and update into a single stabilize function, combine original notify and flush function into rectify function. These will be described in the Implementation section in detail.

II. IMPLEMENTATION

A. Data structure used

- JSON file stores all configs including successor list length, ring size, different kinds of periods for stabilization/fix finger/join retry
- Both finger table and successor lists are saved in 2D list, where each single 1D list has the form [id, ip]
- State machine is saved as dictionary. Logs are saved in lists of [key, value], sorted chronologically; However, if we would like to move data around by its hash key, then we would need to save the logs in 3D list, where the index would refer lists of [key, value].
- Python logging is used to record debug data in both console and on the disk

B. Implementation differs from or not specified in Chord

- The Chord mentioned the usage of successor list for fault tolerance, however, the implementation with successor list wasn't actually specified in the paper.
  - Checking for closest preceding node now checks both successor list and finger tables for the closest preceding node (algorithm 1)
  - Implementation of find successor list which returns the full successor list
  - Stabilization now verifies successor list instead of successor, and it starts by trying from the head of the successor list. After it verifies and establishes the successor to be either first responding successor

or its predecessor, it queries the established successor for its successor list, and append all entries but last to form the updated successor list, however, if the entry go past the current node, it will be ignored.

```
//m is the number of bits, r is the length of successor list
Node Closest_preceding_node (query_id):
    Init res to be pair of (current node id, current node ip)
    For i = m to 1:
        If finger_table[i] is between current node id and query_id:
            Record finger_table in res
            Break
    For i = r to 1:
        If successor_list[i] is a closer predecessor than res:
            Replace res to be successor_list[i]
            Break
    Return res
```

Fig. 1. check closest preceding node

- Stabilize, fix_finger and rectify each has a single thread running the process, where Condition Variables are used to either sleep or wake the thread.
  - Upon joining the Chord Ring successfully, it would notify stabilize to start running
  - After Stabilize verifies and establishes successor list, it would notify rectify for verification
- The joining of first node into the Chord ring wasn't specified in the implementation, thus we solved by following implementation
  - Give the first node an identifier
  - When the first node in Chord Ring receives the first find successor request, which occurs when the 2nd node request to join, it will initiate a thread start joining the first node into the 2nd node.
  - The join request from the first node to the second node will carry an ID of -1, where the receiving server for find_successor will identify it as a special case and return the ID and IP of itself.

```
Void Join(known_node.id, known_node.ip): //id and ip are of the node we try to join
    //handle the special case where the initial node would like to join the ring
    If it's first node in Chord Ring trying to join the 2nd:
        Query the known node with id = -1
    Else:
        Query the known node to find successor for id of itself
    If query returns before timeout:
        Successor_list[0] = query result
        Querry Successor_list[0] for its successor list:
        If successor list result returns before timeout:
            Successor_list = [Successor_list[0],
            append all entries but last(successor_list)]
        Retry join later
    Retry join later
```

Fig. 2. Join function

- Replication, the paper mentions either replication by storing the data associated with key at nodes succeeding the key. While the original algorithm proposes propagating of data to replicas, we implemented with alternative ways:
  - In order to for the distributed systems to sustain after failures of k nodes, we would use a successor list of 2k+1 nodes. We have a quorum system for the replication, where the successor would try to replicate its data to 2k+1 nodes; Only upon finishing replication to majority that the server would reply success to the client
  - Logs are maintained to carry the replicated entries, either for itself or is replicated when it's in the successor list of its predecessor. State machine is a dictionary which carries the results with the application of entries, where it isn't in the critical path for the process and replies.

```
Void Append_but_last(new_successor, its_successor_list):
    Sucessor_list = [new_successor]
    Iterate through s in its_successor_list:
        If s is current node:
            Break
        Else:
            Append it to Successor_list
    If length of Successor_list < r:
        Append (r - length(Successor_list)) * [Null, ""] at the end
```

Fig. 3. Append_but_last function

```
Node Find_successor(id):
    //deal with the special case where first node
    //of Chord trying to join second node
    If request.id == -1:
        Return id and ip of current node
    Elif request.id ∈ (current node id, successor_list[0].id]:
        Return id and ip of successor_list[0]
    Else:
        Pred = Closest_preceding_node(id)
        Return id and ip of Pred.Find_successor(id)
```

Fig. 4. Find successor function

- Authentication: while the original paper proposed to authenticate the data by hashing the key. Additional details need to be paid attention to, the hashed value is first converted to node ID, the node ID need to be confirmed to be between the current node's predecessor and itself so that the current node is correct node to store the information
- Dealing with hash collisions, even though this paper uses SHA1 to achieve consistent hashing. And we can reduce the chance of hash collisions with relatively large m, it's still unavoidable. However, this paper didn't address how to deal with such conditions, we consider the following solutions:
  - Linear probing: we could have the node joined at next available ID. However, not only it requires

modification of the protocol algorithm, it also has clustering issue, and making it inefficient when we try to look up a node since successors are also searched upon hash collisions. Since it increase look up length, we give up this idea.
  - Rehashing: we could rehash the node IP using a different hash function. However, this solution also increases the look up length.
  - Change port number: Since hashing take both node IP and port number, we simply give the node another port number and re-calculate the hash value until there is no hash collision. This solution is adapted since its look up length is lower.
- We referred to How to make Chord correct for correct implementation of Chord, however:
  - Between: it checks whether a given node ID n2 is between node ID n1 and n3:

```
Boolean function between(node id n1, node id n2, node id n3):
    If (n2 == null)
        return false;
    If (n1 == null or n3 == null)
        return true;
    If (n1 < n3)
        return (n1 < n2 && n2 < n3)
    Else
        return (n1 < n2 || n2 < n3)
```

Fig. 5. between function

C. Possible future improvements to Chord

- ID and IPs of the received requests could also be added to finger tables and successor lists when applicable.
  Pros: the stabilization time of Chord successor lists could take as long as (stabilization period + single stabilization process time) * length of successor lists; However, with such implementation, we could shorten the stabilization time.
  Cons: the successor list would need to be synchronized, and frequent change to the successor list might add overhead.

III. Experiment setup and evaluation

We implement our Chord protocol in a recursive style. In the recursive style, each intermediate node forwards a request to the next node until it reaches the successor. The result will be returned back along the calling path. We set the size of identifier circle m to 12 and size of successor list to 2logN where N is the total node number. We implement key replication on top of Chord protocol, copying key to node's successor list.

We develop our protocol locally for easy modification and testing. Final testing is performed on Amazon EC2 using t2.micro. We are running more than one process each server to achieve large amount of nodes. We scale number of nodes from 10 to 50 to observe the performance trend. And the number of virtual nodes on each physical node ranges from 1 to 10. We measure the influence of virtual node on load balance and path length. Client testing is performed locally for easier access and modification. We are testing the performance on load balance, lookup path length, node failure and data replication.

A. Load Balance

Load balance is an important property in distributed hash table(DHT). We first consider the ability of consistent hashing to allocate keys to nodes evenly. Given a Chord Ring with N nodes and K keys, the ideal situation is that each node stores roughly K/N keys.

We consider the Chord network with different number of nodes. We range the number of nodes from 10 to 50, with an increment of 10 nodes. We allocate 10000 keys to the Chord Ring and count the number of keys each node stores. Percentiles of the number of keys per node are plotted in figure 6, including Min, 25%, Median, 75% and Max. In a Chord network with 10 nodes, the mean number of keys each node stores should be 1000 while the max number of keys is around 2200 and 75% is around 1400. In a Chord network with 50 nodes, the mean number of keys should be 200 while the max number of keys is around 900 and 75% is around 250. Increasing the number of nodes in the Chord system will reduce the Max and 75 percentile of the number of keys one node stores, but the relationship may not be linear. We set up another experiment with 50 nodes and range number of keys from 10000 to 100000. The percentile results are plotted in figure 7 where we observe percentiles of number of keys stored on one node scale linearly with the total number of keys. In figure 8, we draw the PDF of number of keys with 50 nodes and 10000 keys.

The reason for the distribution variety is that the node identifiers don't uniformly cover the entire Chord Ring. According to the property of hash function, the number of keys allocated to one node depends on the distance between the node and its predecessor. Longer distance tends to allocate a larger number of keys to one node.

The keys tend to be allocated more evenly with more nodes. However, it is difficult to add extra physical nodes. We implement the Virtual node, associating each physical node with more than one virtual nodes. The identifier of virtual node is calculated using different hash functions on physical node's IP address. Mapping multiple virtual nodes to each real node provides a more uniform coverage of the identifier space. We use 50 real nodes and 50000 keys, associating each real node with different number of virtual nodes, ranging from 1 to 10. Figure 9 shows the percentiles of number of keys per node with different number of virtual nodes for one real node. The Max, 75, 25 and Min percentile move towards the mean when we increase the number of virtual nodes. Without virtual node, the Max percentile is around 4.4 times the mean number, 75 percentile is around 1.3 times the mean number. With 5 virtual nodes per real node, the Max percentile is around 2.8 times the mean number and

75 percentile is around 1.2 times the mean number. With 10 virtual nodes per real node, the Max percentile is around 1.6 times the mean number and 75 percentile is around 1.2 times the mean number. We conclude that associating more virtual nodes with one real node can help improve load balancing.

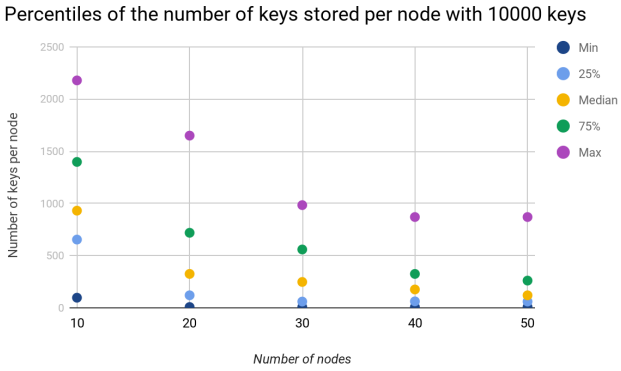

Fig. 6. Percentiles of number of keys per node VS number of keys

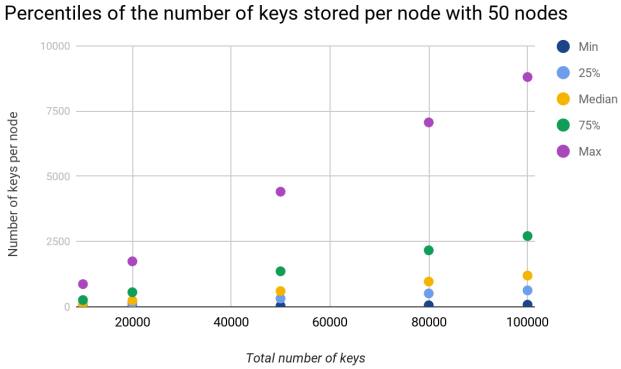

Fig. 7. Percentiles of number of keys per node VS number of keys

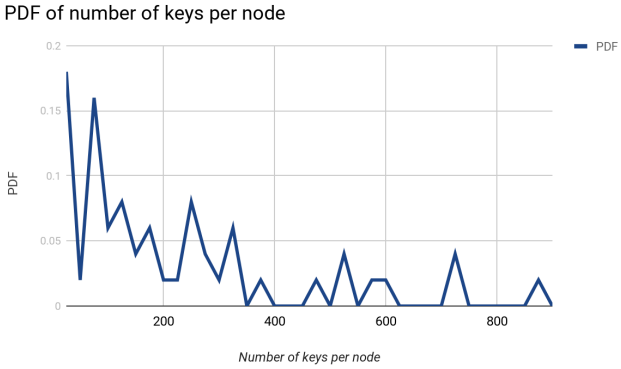

Fig. 8. PDF of number of keys per node

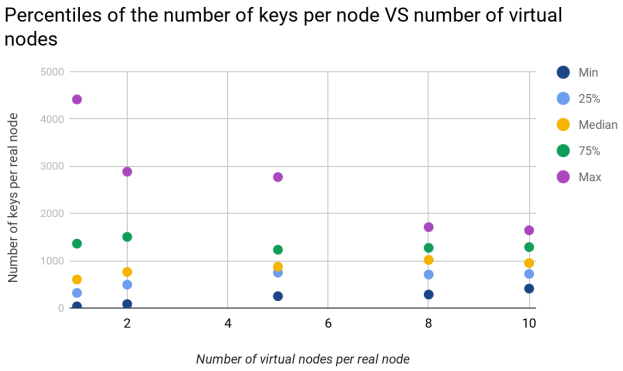

Fig. 9. Percentiles of number of keys per node VS number of virtual nodes

B. Path length

Chord network's performance partly depends on the number of nodes that must be visited to find an identifier's successor. Because we need to find the successor whenever a node joining the network or running a query for a key location. And fix_finger() function calls find_successor() periodically. So the system's performance highly relies on the efficiency of find_successor() function. Assume that gRPC call between servers takes relatively constant time so that the time cost of find_successor() function grows linearly with the path length.

We set up an experiment to get the percentiles of path length with different number of nodes. We range the number of nodes from 10 to 250 and run the find_successor() function for 50000 keys generated randomly. Figure 10 shows the percentiles of path length for different number of nodes. We observe that path length is in O(logN) where N is the number of total nodes. And the mean path length is around *log(N). The time cost for finding the successor for a key grows by log of the number of total nodes. Figure 11 shows the PDF of path length for a Chord system with 250 nodes.

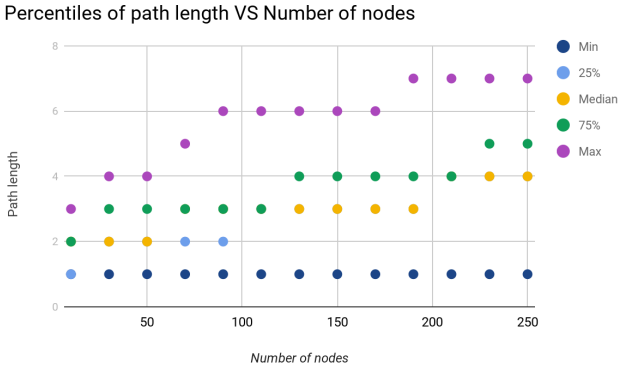

Fig. 10. Percentiles of path length VS number of nodes

C. Stabilization and fix_finger time

When a node joins or leaves the Chord network, the Chord protocol will call stabilization periodically. Each node keeps

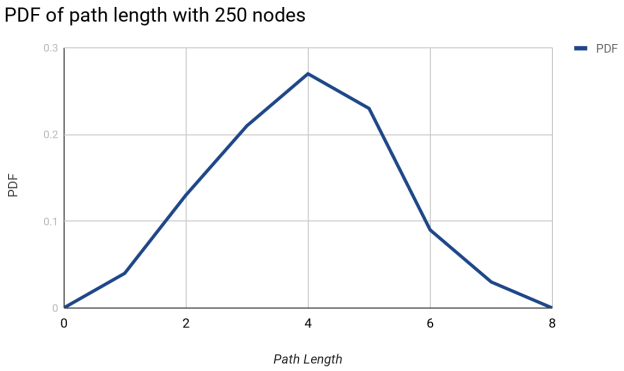

Fig. 11. PDF of path length with 250 nodes

a daemon thread which will call stabilization function every 200 ms. In order to update all elements in the successor list, the stabilization function must be called r times where r is the size of successor list. Figure 12 shows the time for updating the whole successor list when new node joins the network in a 250 nodes Chord Ring. Figure 13 shows the time for updating the whole successor list when a node leaves the network. We observe that the stabilization time grows linearly with the size of successor list.

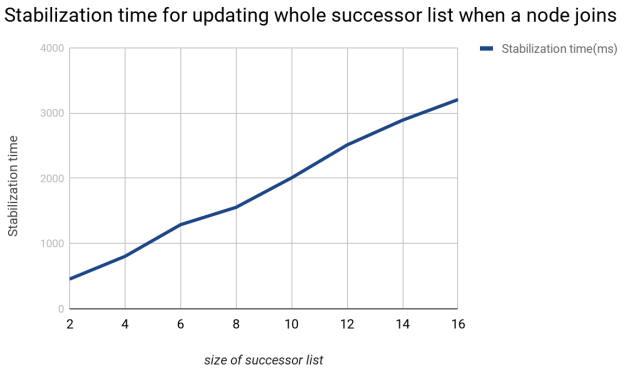

Fig. 12. Stabilization time when a node joins

Each node dynamically updates its finger table by periodically calling fix_finger() function. Fix_finger() function updates one item in finger table at a time. In order to update all items in the finger table, the fix_finger() function must be called at least m times where m is the size of finger table. Each node keeps a daemon thread calling fix_finger() function every 100ms. Figure 14 shows the time for updating the whole finger table in a Chord Ring with 250 nodes. We observe that the time for updating whole finger table grows almost linearly with m, the size of finger table.

D. Node failures

We want to evaluate the influence of node failures on lookups. We consider a Chord network of 250 nodes with

Fig. 13. Stabilization time when a node leaves

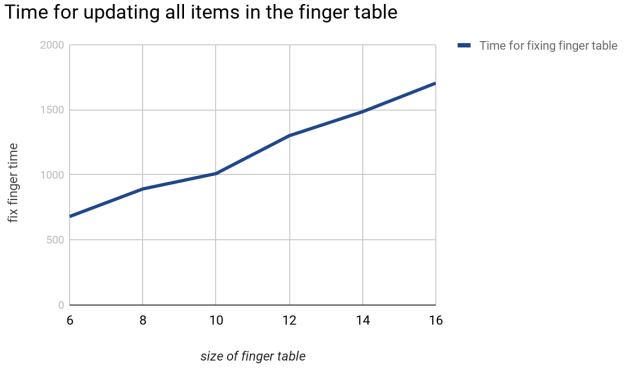

Fig. 14. Time for updating finger table

successor list of size 16. By implementing the Chaos Monkey feature, each node is made to fail with probability p. Firstly we wait for the 250 nodes becoming stable, then we enable the Chaos Monkey feature. We conduct lookup query for 10000 keys generated randomly and count the path length. Figure 15 plots the percentiles of path length for different fractions of failed nodes. Massive failure on Chord network will not severely influence the path length required for key lookups. When a network of 250 nodes suffers a failure with 50% of all nodes, the percentiles of path length increase approximately only by one. We also count the number of timeouts during the lookups. There are almost no timeouts even with 50% failure rate. We conclude that the Chord's performance will not be severely influenced under the circumstance of massive failure and it can always provide correct lookup service.

E. Data replication

When a client sends a PUT request to any node in the Chord network, it firstly finds the successor for that key, then the successor will not only store the key locally but also copy the key to all nodes in its successor list. After receiving responses from more than half the nodes in the successor list, the successor node replies to the client for a successful PUT. Using this replication method will increase

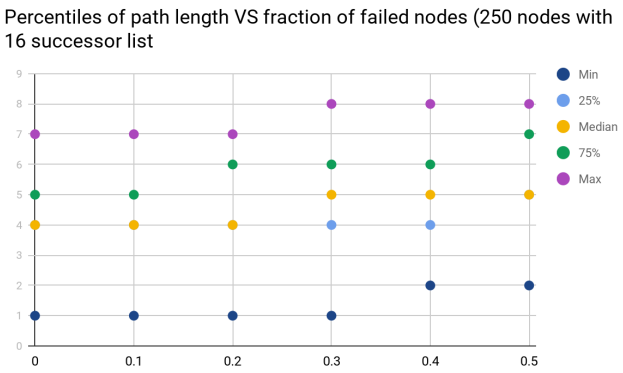

Fig. 15. Percentiles of path length VS fraction of failed node

the overhead for adding a key to the Chord network. But it can ensure that there won't be data loss with k nodes failure by using successor list of 2k+1. We set up a Chord network with 250 nodes and range the size of successor list from 2 to 16. Figure 16 shows the replication time for different length of successor list. The replication time grows sub-linearly with the length of successor list.

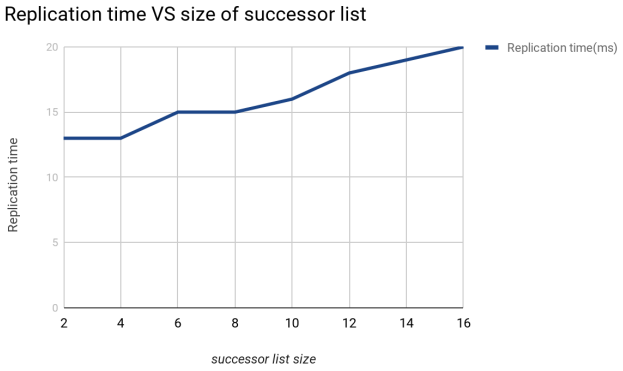

Fig. 16. Replication size VS size of successor list

IV. Chord Correctness

To prove the correctness of the Chord protocol, we must obey the following rules:

There must be a ring including a non-empty set of ring members.

There must be no more than one ring which means every ring member is reachable from any ring member following its successor.

The nodes must follow the identifier order in the Chord Ring.

The Chord Ring must be reachable from any node joining the network.

The Chord network is under continuous changing. There are two main events leading to network change. The first one is a node joining the network and the second one is a node leaving the network, either due to failure or voluntary.

Figure 17 shows the procedure when a node becomes part of the Chord Ring. A gray circle marks the pointer updated by an operation and dotted arrows are predecessors. When a node joins the network, it contacts any existing node in the network to get its successor and successor list. When a node stabilizes, it learns its successor's predecessor. Each node periodically calls stabilize operations. After stabilizing, a node notifies its successor which causes the notified node to execute a rectify operation. These operations guarantee that the Chord network will become stable after node joining the network. A node ceases to become a member when it fails, either because of node failure or voluntary leave. When a node leaves the Chord network, there will be a gap in the network. Stabilization will repair the gap and the network becomes stable again.

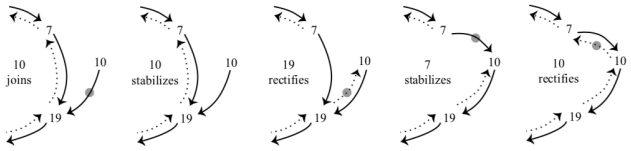

Fig. 17. Procedure when a node joins the Chord network

V. Conclusion

Chord provides a scalable peer-to-peer lookup protocol. In a Chord network with N nodes, it achieves consistent hashing, each node stores information for only O(logN) of other nodes and the lookups only take the path length of O(logN). Our implementation of Chord is proved to be correct, load balancing, scalable and available.

The contribution of this paper is that initialization of Chord Ring at size 1 is now made possible through changing the algorithm; replication of (key, value) pair to successor lists of length 2k + 1 could successfully live upon the failures of k nodes; hash collision solutions are handled by choosing a new port number, which could maintain O(log N) path length.

Evaluation of our implementation of Chord achieves satisfactory result:

- Load balancing: With our randomly generated (key, value) pairs, each node carries roughly $(1 + \epsilon)K/N$ pairs of data.
- Scalability: the look up length close to O(log N), Put/Get response time is sublinear to the number of nodes.
- Safety: The length of successor list being 2k+1; Even with failures of k nodes in one single successor list, the data is still safe in our implementation.
- Availability: Our Put/Get could always return either the successful result or a hint of ID/IP of node that contains the data.
- Correctness: In addition to theoretical proof, multiple cases are designed. The Chord protocol can always put/get correctly in changing environment.

In terms of load balance, each node in the Chord network tends to store K/N keys where K is the total number of keys and N is the number of nodes in the system. The network becomes more balanced as we add more nodes. We introduce virtual nodes, associating each real node with more than one virtual nodes can improve load balancing.

In terms of scalability, the Chord network can be easily scaled because each node stores information about only O(logN) about other nodes and lookups take path length of O(logN) where N is the total number of nodes in the network.

In terms of availability, lookups can be executed given any node in the network and the Chord system is available even under massive failure.

Future work could include increase of network speed on AWS servers since network speed is our current bottleneck during replication. GO or other languages are desired to increase our server side performance.

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] *Zave, Pamela. "How to make chord correct (using a stable base)." CoRR, abs/1502.06461 (2015).*

[3] *Karger, David, et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web." STOC. Vol. 97. 1997.*

[4] *Van Renesse, Robbert, and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability." OSDI. Vol. 4. No. 91104. 2004.*

[5] *gRPC*, https://grpc.io/