ChordDFS: A Distributed File System Built on Chord

Alex Scarlatos and Wendy Zheng

May 2018

1 Introduction

Distributed systems are a way of removing excessive load from any one location, and using as many resources as available to achieve a task. In the case of file systems, we can avoid having a single server for every file, allowing traffic to be evenly distributed between multiple servers and requiring less storage space on each server.

We can improve this paradigm by allowing the network to be dynamic, where a large number of nodes can enter the network at any time and participate in hosting files. In the real world, this could be regular users who would donate a small amount of disk space in return for being able to access files on the network. This allows for the network to grow to almost any size, as long as a few obstacles can be overcome. First, any given client has to be able to retrieve any file in a reasonable time. Second, one must account for the unreliability of these nodes, and make sure that even when they join and leave the network, files are not lost from the network and remain evenly distributed.

We use Chord [1], a peer-to-peer protocol for distributed networks, as the backbone of our system, ChordDFS. Chord enables any node in the network to find the location of a key/id, in our case a file, quickly and without knowing where it is stored. It also ensures that files are evenly distributed throughout the network. We implemented a Chord process in Python that can be run on any independent node capable of file and network IO.

The primary focus of this paper will be to determine how network unreliability can be handled in a distributed file system built on Chord. We will describe a custom algorithm for recovering from unexpected node behavior, and evaluate how well our system works under different settings.

2 System Description

2.1 Chord

Chord uses consistent hashing to assign keys to nodes, which tends the balance the load or data that any given node in network has to be responsible for. The nodes in the network that are running Chord are also *connected* in a ring, known as the Chord Ring. Operations on the nodes in the network and on the Chord Ring are what guarantees the O(logN) look up time of keys in the size of the network.

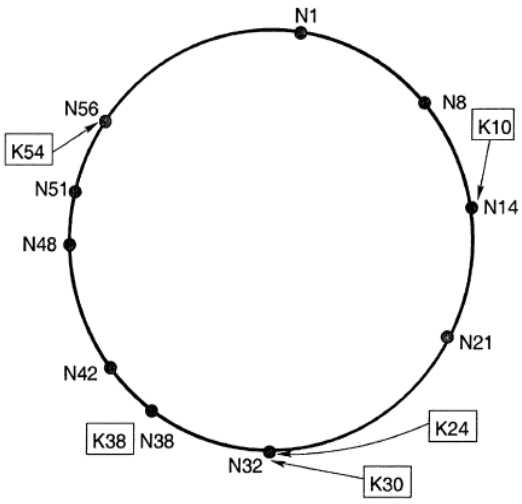

Figure 1: Chord Ring

We will use Figure 1 to illustrate some of the concepts we will be using throughout the paper.

1. Chord Ring: the ring of nodes that are running the chord protocol. The size of the ring is 2^m. For m=6, we have the Chord Ring as shown in Figure 1. In our implementation, m is determined based on the number of nodes in the network (m = ceil(logN) + 3) to decrease the chance of id collision. The ids of these nodes are obtained by running a hashing algorithm on the IP address of a given node, and taking the modulo of the ring size. If the hashing algorithm is consistent, the ids should be evenly distributed across the ring. The time required for the creation of this ring will be defined as the stabilization time.

2. Successor node: the successor node of any given node is the node directly following it clockwise on the Chord Ring. For example, the successor of N1 will be N8 in Figure 1. The *findSuccessor* method finds this successor for a given node by propagating a message around the ring until the requested id is found.

3. Predecessor node: the predecessor node of any given node is the node directly before it counter clockwise on the Chord Ring. For example, the predecessor of N1 will be N56 in Figure 1.

2.2 Chord Implementation

Our system utilizes mininet to generate a network of nodes that run the Chord protocol [2]. Our current network topology consists of having a single switch in the center and all other nodes in the network are connected through this switch. This topology is used as an abstraction for the real world, where hosts are connected together through the Internet. In our system, we use the UDP transport protocol to allow our nodes to communicate with

one another instead of TCP due to simplicity. However, we do realize that TCP can also be used to ensure that file transfers and insertions are not lost.

2.2.1 Tracker

A "tracker" node is used as a point of entry into the network. Chord requires that when a node wants to join a network, it must send a *findSuccessor* request to any other file in the network, which would be the tracker in this case. In a real world scenario, trackers could be found using DNS. Trackers would be highly monitored servers with a very low fail rate, but we would only need a few of them. In our implementation, we experimented with just one tracker.

Trackers are also used to keep track of every file that is in the network. They do not know the locations of the files themselves (since this could change as nodes join and leave), but know the name and id of every file in the network. Files must be inserted into the network through a tracker, and trackers could communicate with each other to construct a full file list. When a tracker receives a file to insert, it discovers the destination node through the Chord protocol, and transfers the file to that node.

2.2.2 Client

A client application can be run to interact with the ChordDFS system. In particular, the client talks to a tracker node, who acts as the gatekeeper of the Chord Ring, in order to initiate various operations. The user running the client can run the following operations:

- **get file**: retrieve a given file from the network
- **insert file**: insert a given file into the network
- **list**: list all the files that are available on the network

Error handling is also implemented for these operations; i.e. when a get request fails, the client retries up to 3 times before it gives up and tells the user that the file does not or no longer exists on the network. In addition, the client application can also be run with a script. This will be useful for network administrators who have a set list of files to retrieve or insert into the network. They can pass the operations through the client, which will then run them automatically.

2.3 Configurations

The following configurations can be tuned to test the ChordDFS implementation:

- control port: what port the chord node servers are listening at
- tracker_node_ip: the IP the tracker node is located at
- num_replicates: the number of replications per file node, we left this at 3 for the experiments

- refresh_rate: how often (in seconds) the network checks for updates, we left this at 1 for the experiments
- leave_join_prob: the probability a node will (gracefully) leave or join the network every time it refreshes
- fail_prob: the failure probability of a given node every time it refreshes
- client_rate: how often (in seconds) a client sends its next request to the tracker node

2.4 Node Join and Departure

When a node joins or gracefully leaves, the network transfers files properly, and no losses should occur. It transfers its files to its successor before informing its successor and predecessor that it is leaving. However, when a node fails, this is more representative of a computer crashing or losing internet connection. In this case, it cannot inform any other nodes of its departure, and its hosted files are lost.

2.5 File Replication and Recovery Under Node Failure

To account for the possibility of nodes failing unexpectedly, files must be replicated throughout the network so that they can be recovered when their hosts fail. We created a custom algorithm to recover from this, which is different from the one proposed in the Chord paper. The authors proposed using a successor list, meaning that a file on any given node should be replicated on its next k successors in the network. The problem with this is that every node would have to internally maintain information about many other nodes in the network. Our solution does not require any nodes to know any additional information or state.

Files are replicated on insertion into the network. Instead of getting one id, every file its hashed k times and thus gets k different ids. The file is then inserted regularly at each of those ids. Assuming consistent hashing, each of these copies should be sent to a different node in the network. Now, when a node dies, there is a high probability that someone else in the network has a copy of each of the files it was responsible for. The problem is that now there will be one copy less of these files in the network, and we want to reinsert them to ensure that a copy exists for each hash. We do this by detecting the death of the node.

A node knows when its successor dies by detecting when it is no longer receiving responses from it. When a node detects its successor's death, it will send an alert to a tracker, sending its own id and its successor's id. The tracker gathers the names and ids of all files in the network, examines them, and finds all files that should have gone to the now dead node, according to the Chord protocol. For each of these files, the tracker uses the various ids to find a different node on the network that hosts that file (using *findSuccessor*). The tracker tells the alternate host to reinsert the file at the lost id, which it can do since it has a copy. When the file is reinserted, a different node will now be responsible for the id that was lost because the network will have stabilized.

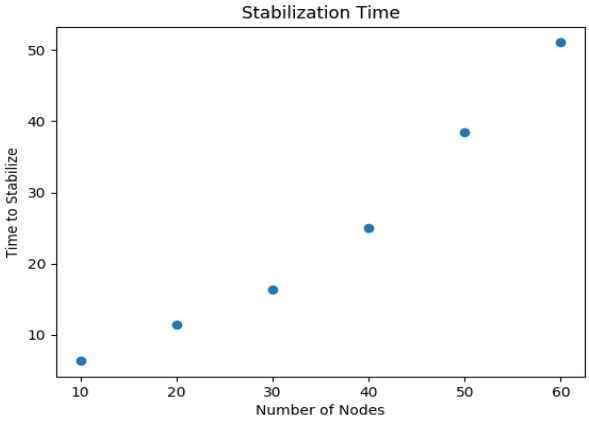

Figure 2: Network Stabilization Graph

3 Evaluation and Results

3.1 Network Stability

Figure 2 shows the stabilization time of our Chord Ring plotted against the number of nodes in the network. As you can see, the trend is polynomial, as expected when using the *findSuccessor* algorithm for stabilization. We did not use Chord's notion of "finger tables" in our implementation, which would have resulted in a more linear curve. Unfortunately, while we did implement finger tables, we did not have enough time to test it thoroughly, therefore we will not include its results in our evaluation. However, we claim that the general trends shown in the following sections would be similar for finger tables and without. This is because using finger tables only increases the speed of which *findSuccessor* correctly finds the nodes that are responsible for a given id and thus does not affect correctness.

3.2 File Distribution

The evenness of a file distribution across a network can be measured by examining the number of files hosted on each node. Ideally, the number would be similar on each node, and we can quantify this by taking the variance of the distribution. The following table shows the variance of the distribution of the file keys across all nodes of the network measured using different configuration parameters. You can see that the minimum variance is 1.63, the maximum variance is 4.38, and the average variance is 2.13. This means that our distribution of file keys is actually quite consistent, which is what we want.

Fail Rate	Leave/Join Rate	Client Op Rate	File Distribution Variance
10^{-2}	10^{-4}	1	1.73
10^{-2}	10^{-4}	0.1	1.69
10^{-3}	10^{-4}	1	1.73
10^{-3}	10^{-4}	0.1	**1.63**
10^{-4}	10^{-3}	1	1.81
10^{-4}	10^{-3}	0.1	**4.38**
10^{-4}	10^{-4}	1	2.23
10^{-4}	10^{-4}	0.1	1.73
0	10^{-4}	1	2.23
		Avg Var	2.13

3.3 Performance

3.3.1 Metrics

We performed several experiments to discover the effects of different parameters on

1. the amount of time it took to receive network responses (in hops)

2. the success rate of requests

We define a request as unsuccessful if at any point it is lost in the network. This could happen if it is being processed at a node that dies, or if it is sent to a node that dies before its death is detected. While this problem could be alleviated by using TCP connections, it is interesting to look at regardless, since it provides some insight into the stability of the network. It is also a good indicator of how much non-congestion related loss would be experienced for TCP in deployment.

3.3.2 Client Request Rate

Figures 3 and 4 show that the rate at which clients request files has a significant impact on how long it takes to receive a request. We can see that when the rate is low (meaning in this case the client sends successive requests very quickly), the average number of hops for a response tends to be lower. This trend holds for both low and high failure rates of nodes in the network.

We can also see in Figure 5 that a longer gap between successive client requests actually reduces success rate. Under low loss, all requests were returned, so we did not show this case.

A possible explanation for quicker client requests reducing response time is that these requests were made before the network got a chance to stabilize. While the network is unstable, there is a chance that the requested files are closer than they will be later, making the time to find them faster.

The reason that quicker requests improve success rate is probably similar. Files were inserted and retrieved before many nodes had a chance to fail. This indicates that bursty traffic actually performs better than regulated traffic on our network.

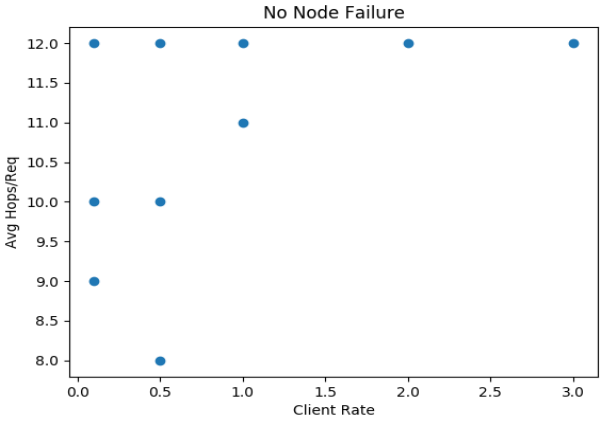

Figure 3: Effects of Client Rate on Performance without Node Failure

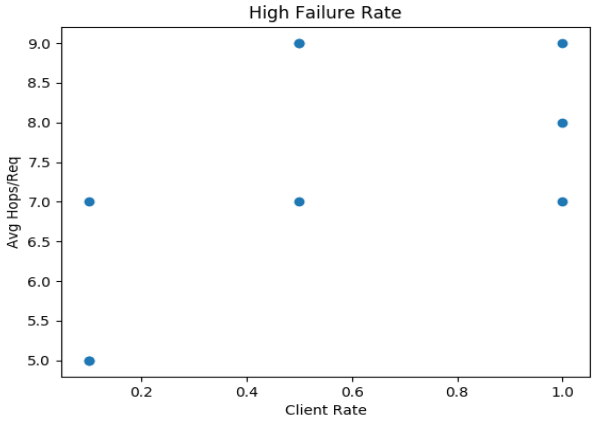

Figure 4: Effects of Client Rate on Performance with High Node Failure

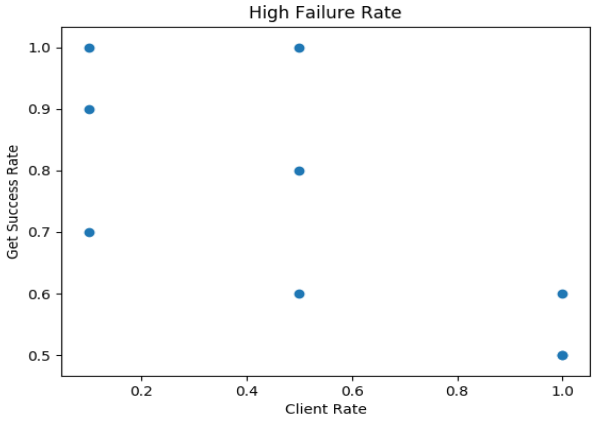

Figure 5: Effects of Client Rate on Client Request Success with High Node Failure

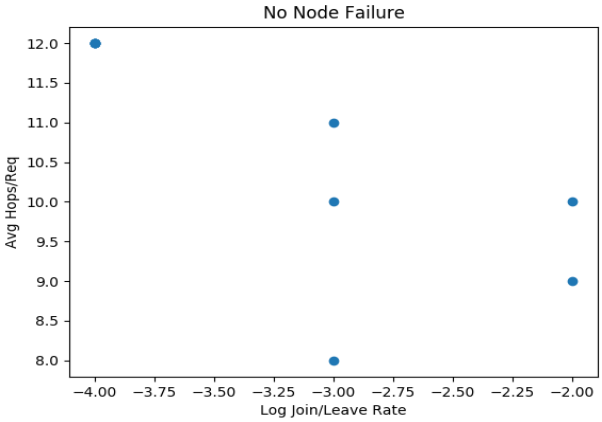

Figure 6: Effects of Node Join/Leave Rate on Performance without Node Failure

3.4 Node Join/Leave Rate

In our experiment, we tested the effects of node join and leave rates on how fast a client operation is answered when varied on no failure and high failure rates (1%).

In Figure 6, you can see that we plotted the log of the join and leave rate against the average number of hops per request given no fail rate. Our results show that the smaller the leave and join rate, the higher the average number of requests. However, we also note that the variability in the average number of hops per requests is also higher when there is a lower leave and join rate. The higher average number of hops despite low leave and join rate can be attributed to the fact that there are more nodes in the network, and none of those nodes leave, thus maintaining the size of the network. Since the size of the network is large, it requires more messages to be propagated from node to node until the correct successor that is responsible for a given file is found.

In Figure 7, we plotted the log of the join and leave rate against the average number of hops per request given high fail rate. We see high variability in the number of hops it takes to answer a client request for all variations of join and leave rates. This high variability is most likely caused by the contending effect of the failure rate and the join and leave rates on the nodes. Nonetheless, aside from high variability, there are no other obvious trends.

In Figure 8, we plotted the log of the join and leave rate against the average get operation success rate. We see that there is high variability in the results, which can be attributed to high node failure. However, can extrapolate that high leave and join rate tends to decrease success rate while low leave and join rate has the opposite affect. The variance is largest when we have a leave and join rate in the middle.

3.5 Node Failure Rate

Figure 9 shows that a high failure rate actually decreases the amount of time taken for each request on average. This could be due to several factors. It could be that the size of the network temporarily decreases, which causes the paths to files to become shorter. It could also be that requests that take longer are more likely to become lost by a failing node, so only

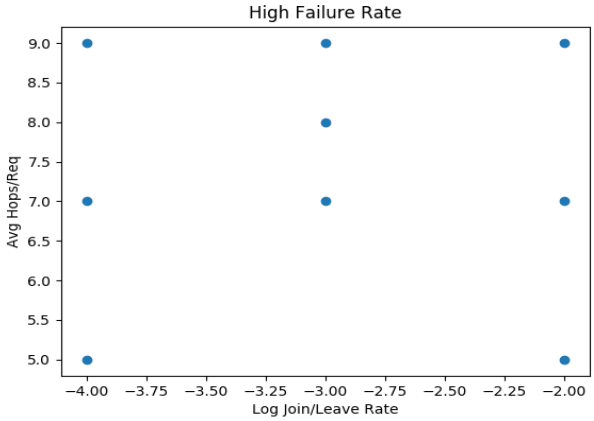

Figure 7: Effects of Node Join/Leave Rate on Performance with High Node Failure

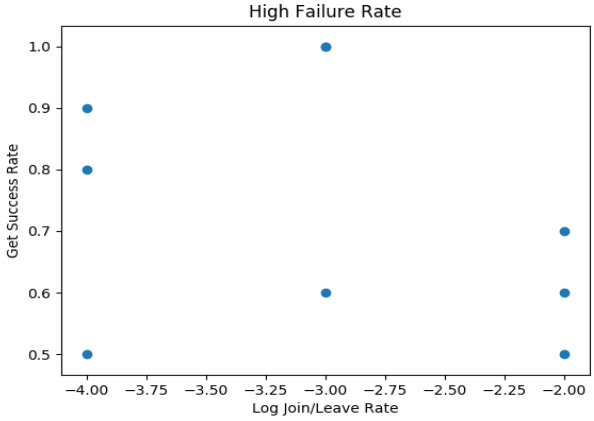

Figure 8: Effects of Node Join/Leave Rate on Client Request Success with High Node Failure

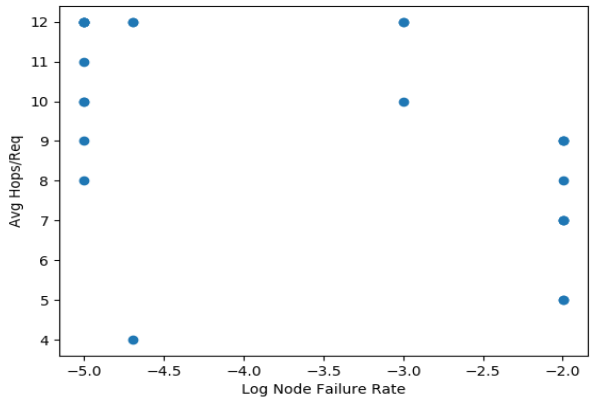

Figure 9: Effects of Node Failure Rate on Performance

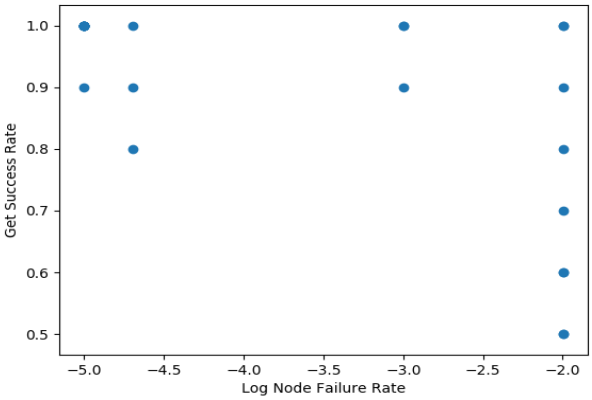

Figure 10: Effects of Node Failure Rate on Client Request Success

the shorter requests are returned. In the future, we would like to investigate this further.

We can clearly see in Figure 10 that node failure can have an adverse effect on successful file retrieval. However, what is surprising is that under high failure rate, it is still possible to achieve a very high success rate. But the probability of a low success rate still increases. This indicates that high failure rate introduces variability into the network, but is not guaranteed to cause losses.

4 Conclusions

The Chord protocol provided the foundation of our implementation of a distributed file system. We believe that our system is a good proof of concept and that many of the trends we discovered along the way can be extended and applied on a real world distributed file system. Since our system is highly configurable, it can also be a test simulation bed to explore the effects of different configurations on a given network.

One configurable parameter in the system that we believe is especially important is the

failure rate. Failure rate in the network has both benefits and downsides. A node failing causes the network to restabilize, which we discovered increases the probability of files being retrieved quicker. However, a high failure rate also causes requests to be lost. While these losses could be recovered using a connection-based protocol in deployment, they are indicative of performance issues that protocol would face.

At our highest tested failure rate, any given node should be expected to fail every 100 seconds (a 1% chance of failure at a 1 second refresh interval). This is a very high frequency for most realistic settings, where most nodes would exit the network gracefully. But in some networks where connections can be lost unexpectedly, such as wireless ones, we may experience high failure rate. In these networks, ChordDFS could still be deployed, but there may be significant slowdown.

5 Future Work

Mininet restricted the number of nodes we could use in the network at any given time, and to truly test the practicality of our system we would want to experiment with a realistically-sized network (10's of thousands of nodes). Additionally, we would also want to measure the impact of using different numbers of trackers on file distribution and client response time.

We noticed that often nodes ended up with the same ids. This indicates that we could have used a better hashing technique, and more research would need to be done on how to avoid collisions with higher probability.

Additionally, in our implementation we only used one tracker node as the point of entry into the network for any given client. Realistically, we would want the tracker node to be the node that is closest to the client node. This can be modeled in our system by having multiple tracker nodes scattered across the Chord Ring. In this case, both the average response time and the average number of hops should be lower.

6 References

References

[1] Chord Paper

[2] Mininet

[3] ChordDFS GitHub Repo