ChorDNS: Efficiently Hashed DNS Resolution

Tarush Abhaya Jain, Dheeraj Ramchandani,
Benjamin Michalowicz, Ghanendra Piplani
https://github.com/ghanendrapiplani/ChordDNS

December 2020

Abstract

Chord is a peer-to-peer DNS service. What this means is that the performance made by the service and administration of a given hierarchy. Name-server administration errors are a common point of failure in a hierarchical DNS server. One way to keep the DNS Hierarchy without redesigning the entire DNS hierarchy is through the use of DNSSEC, though this can prove to be inefficient as the distance between the original host and the final location of a hostname increasesIn addition, DNS with a traditional hierarchy has the added issue of returning a large number of negative answers or no answer at all, all because of incorrectly configured name servers or incorrect NS records. In our project, we can show that Chord, while working with multiple record types, can not only allow for better load balancing in a distributed network, but also feature better return statistics and faster DNS query times than a traditional DNS hierarchy (Root Servers → TLD Servers → Authoritative Name Servers → Local DNS servers). We compare our results with other DNS tools such as `dig`, python's `DNSPython` library, and `nslookup`, along with A, MX, and NS records. The rest of the report will continue as follows: Section 1 will explain previous/related work. Section 2 will explain how we obtained our data and used it for our experiments. Section 3 will break down Chord. Section 4 will break down and explain DistAlgo. Section 5 will break down our experiments. Section 6 will show and discuss our results. Section 7 will draw our conclusions from the project, and section 8 will examine future work.

1 Previous/Related Work

1.1 Freenet

Freenet is a peer-to-peer platform for censorship-resistant communication. It uses a decentralized distributed data store to keep and deliver information, and has a suite of free software for publishing and communicating on the Web without fear of censorship. However, Freenet does not assign responsibility for documents to specific servers; instead, its lookup queries take the form of searches for cached copies. This allows Freenet to provide a degree of anonymity, but prevents it from guaranteeing retrieval of existing documents or from providing lowbounds on retrieval costs. Chord does not provide anonymity, but its lookup operation runs in predictable time and always results in success or definitive failure.

1.2 Globe

The Globe Distribution Network, or GDN, is an application for the efficient, worldwide distribution of free software and other free data. The GDN can be seen as an improvement to anonymous FTP and the World Wide Web due to its flexibility and extensive support for replication. The global state of the dis-tributed object is made up of the state of the semantics sub-objects in its local representatives. The Globe system handles high load on the logical root by partitioning objects among multiple physical root servers using hash-like techniques. Chord performs this hash function well enough that it can achieve scalability without also involving any hierarchy, though Chord does not exploit network locality as well as Globe.

1.3 Plaxton Algorithm

A simple randomized access scheme that exploits locality and distributes control information to achieve low overhead in memory. Its main main advantage over Chord is that it ensures, subject to assumptions about network topology, that queries never travel further in network distance than the node where the key is stored. Chord, on the other hand, is substantially less complicated and handles concurrent node joins and failures well.

1.4 Grid Location System (GLS)

GLS is a location service which provides location information for a node in a mobile ad hoc network. GLS provides a mechanism for a node to track the location of other nodes in the network topology. GLS is a location service that is built upon a number of location servers distributed throughout the network. There are three main activities in GLS: location server selection, location query request and location server update. GLS relies on real-world geographic location information to route its queries, which may be a caveat if servers move from location to location; Chord maps its nodes to an artificial one-dimensional space within which routing is carried out by an algorithm similar to GLS.

2 Data gathering

Needing a massive dataset to simulate DNS querying and resolution, we first turned to "Rapid7 Open Data" for A, MX, and NS records. However, given the volume of data, and the large number of hostnames and websites that arose from intermediate steps in queries, we turned to a simpler approach. Using the dataset found in [4], we took each of the one million sites, and parsed a subset of each one's A, MX, and NS records, filtering out empty and negative records to house 400,000 records of each type. Specifically, if A, MX, and NS records could be found for a given website, it was added into our database. If one of the queries resulted in an error, it was discarded and we moved onto the next domain name.

3 Chord

3.1 Chord overview

The traditional setup for DNS queries is a tree-based hierarchy, primarily organized into thirteen root servers, top level domains (TLD's), authoritative name servers, and local DNS servers. Using a tool such as `dig` in a Linux environment incurs iterative querying across this hierarchy. The further the distance from the host/starting point, the longer a query may take, and the more likely it will return empty or with an error. Chord is a decentralized, distributed lookup protocol in the form of a hash table spread across different nodes (servers). A hierarchical setup can query only one node at a time; with a distributed and decentralized setup of DNS nodes, one can answer a query in $O(log_2\ N)$ time.

3.2 Distributed Hash Table

A distributed hash table (DHT) is a distributed system that provides a lookup service similar to a hash table: key-value pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. The main advantage of a DHT is that nodes can be added or removed with minimum work around re-distributing keys. Keys are unique identifiers which map to particular values, which in turn can be anything from addresses, to documents, to arbitrary data. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failure. DHTs form an infrastructure that can be used to build more complex services, such as anycast, cooperative web caching, distributed file systems, domain name services, instant messaging, multicast, and also peer-to-peer file sharing and content distribution systems.

DHT's emphasize the following properties:

1. Autonomy and decentralization: the nodes collectively form the system without any central coordination.
2. Fault tolerance: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.
3. Scalability: the system should function efficiently even with thousands or millions of nodes.

Some implementations of DHT's include:

- As Protocols:
    1. Apache Cassandra
    2. Content Addressable Network
    3. Chord
    4. P-Grid
    5. Pastry
    6. Tapestry
    7. Voldemort

- As Applications
    1. FAROO
    2. Freenet
    3. GNUnet
    4. I2P
    5. Retroshare
    6. Twister
    7. YaCy

The four original DHT's are:

1. CAN - Content Addressable Network. A CAN node maintains a routing table that holds the IP address and virtual coordinate zone of each of its neighbors. A node routes a message towards a destination point in the coordinate space. The node first determines which neighboring zone is closest to the destination point, and then looks up that zone's node's IP address via the routing table

2. Tapestry - Tapestry's algorithm comes with an API and Routing model where each node is assigned a unique nodeID uniformly distributed in a large identifier space. Tapestry uses SHA-1 to produce a 160-bit identifier space represented by a 40 digit hex key. Application specific endpoints GUIDs are similarly assigned unique identifiers. NodeIDs and GUIDs are roughly evenly distributed in the overlay network with each node storing several different IDs. Tapestry provides an overlay routing network that is stable under a variety of network conditions. This provides an ideal infrastructure for distributed applications and services.

3. Pastry - What sets Pastry apart is the routing overlay network built on top of the DHT concept. This allows Pastry to realize the scalability and fault tolerance of other networks, while reducing the overall cost of routing a packet from one node to another by avoiding the need to flood packets. Because the routing metric is supplied by an external program based on the IP address of the target node, the metric can be easily switched to shortest hop count, lowest latency, highest bandwidth, or even a general combination of metrics.

4. Chord.

3.3 Chord Protocol Details

Nodes and keys are assigned an m-bit identifier using consistent hashing. The SHA-1 algorithm is the base hashing function for consistent hashing. Consistent hashing is integral to the robustness and performance of Chord because both keys and nodes (in fact, their IP addresses) are uniformly distributed in the same identifier space with a negligible possibility of collision. Thus, it also allows nodes to join and leave the network without disruption. In the protocol, the term node is used to refer to both a node itself and its identifier (ID) without ambiguity. So is the term key.

Using the Chord lookup protocol, nodes and keys are arranged in an identifier circle that has at most 2^m nodes, ranging from 0 to $2^m - 1$. m should be large enough to avoid collision. Some of these nodes will map to machines or keys while others will be empty.

Each node has a successor and a predecessor. The successor to a node is the next node in the identifier circle in a clockwise direction. The predecessor is counterclockwise. If there is a node for each possible ID, the successor of node 0 is node 1, and the predecessor of node 0 is node 2m̂ - 1; however, normally there are "holes" in the sequence. For example, the successor of node 153 may be node 167 (and nodes from 154 to 166 do not exist); in this case, the predecessor of node 167 will be node 153.

The concept of successor can be used for keys as well. The successor node of a key k is the first node whose ID equals to k or follows k in the identifier circle, denoted by successor(k). Every key is assigned to (stored at) its successor node, so looking up a key k is to query successor(k).

3.4 Lookup Details

Since Chord used a Distributed Hash Table(DHT), the lookup model is similar to a key-value based hash table where any node can retrieve a value based on a given key where the advantage lies in adding or removing nodes with minimum work around re-distributing keys.

Chord's two main types of lookups are:

1. Basic - The core usage of the Chord protocol is to query a key from a client (generally a node as well), i.e. to find `successor(k)`. The basic approach is to pass the query to a node's successor, if it cannot find the key locally. This will lead to a $O(N)$ query time where N is the number of machines in the ring.

2. Fast - To avoid the linear search above, Chord implements a faster search method by requiring each node to keep a finger table containing up to m entries, recall that m is the number of bits in the hash key. The i^th entry of node n will contain successor$((n + 2^{i-1}) \ mod \ 2^m$.The first entry of finger table is actually the node's immediate successor (and therefore an extra successor field is not needed). Every time a node wants to look up a key k, it will pass the query to the closest successor or predecessor (depending on the finger table) of k in its finger table (the "largest" one on the circle whose ID is smaller than k), until a node finds out the key is stored in its immediate successor. With such a finger table, the number of nodes that must be contacted to find a successor in an N-node network is $O(logN)$.

3.5 Hashing Function and Collision Avoidance

Originally in the paper, SHA-1 has been used to implement the chord network's fingers, it produces a 160-bit message digest, we observed it leads to significant amount of collisions , x in our case, which is a reason of concern in today's time when there are millions of machines connected to the internet and these collisions could exponentially increase, consequently causing a decrease in performance. Thinking on these lines, we started researching possible solutions to this issue and found that if we replace the SHA-1 algorithm with the newest suite of SHA called the SHA-3, the collisions in our dataset came down to 0 and we can easily extrapolate that the collisions in real world environment would also be comparatively limited, thus being able to preserve the expected performance of Chor-DNS.

4 DistAlgo

DistAlgo is, literally, a language for distributed algorithms. It has four main components: 1. distributed processes and sending messages, 2. Control flows and receiving messages, 3. High level queries of message histories, 4. Easily configurable. High-level queries are particularly helpful for understanding distributed algorithms at a high level, and Python, our language of choice, allows for this to work out, while allowing DistAlgo's constructs to be more declarative (e.g. supporting tuple patterns). Processes are defined as classes extending the "Process" class, with its body containing its own set of handler and method definitions. Special methods such as setup()and run() are set up, with the former initializing a parameter "v" and allowing additional fields to be defined as needed. [6]

Process creation is done by defining a new processes "n" at a set of particular nodes. In abstract terms this is represented as "n new p at node_exp", with node_exp being the set of nodes. In python, this setup, and the setup() and run() methods are represented as follows: [6]

- (initialization) new(p, num=n, at=node_exp)
- (set up for one or more processes in "pexp") setup(pexp, (args))
  - Note: A trailing comma is needed if "args" is a single argument
- (run a set of processes) start(pexp)

Synchronization is done through a call to await(boolExpr), waiting for "boolExpr" to become true. Await statements must be preceded by yield points for handling messages while waiting, because if a yield point is not explicitly defined, all message handlers can be executed at this point. [6] In Python, sending messages to one or more processes is straightforward, as is message reception:

- Sending: sent(msgExpr, to = ProcExpr) ; (msgExpr, ProcExpr) in sent
- Receiving: received(msgExpr, from_=ProcExpr) (msgExpr, ProcExpr) in received

Note: Receiving of messages, or handling them specifically, is done with a user-created "receive" method:

- `def receive(msg = mexp, from_ = pexp, at = (`l_1 `, ..., `l_j `)):`
  `handler_body`

Patterns can be used to match messages in the "sent" and "received" sets. Constant values ('ack', for example) or previously bound values must be specified with the "=" prefix. Underscores ("_") are a match-all expression. For example, receive('ack', =n,_) from a matches every triple in received where the first two components are 'ack', and the value of n, and binds "a" to the sender. Python syntax dictates that the "=" sign is replaced by "_", as the former is not allowed in Python variable names. [6]

Configuring channels is equally straightforward. Three options for channel configuration currently available are `fifo`, `reliable`, and `reliable, fifo`. If any option is specified, TCP is used for process communication, with UDP being the default. In Python, a call to configure the channels is represented as follows:

- `config(channel = 'fifo')`
  `# Can also be 'reliable' or 'reliable, fifo'`

Similarly, with message handling, one can either currently choose to handle one message at a time, or all of them at once: [6]

- `config(handline = 'all')`
  `# config(handling = 'one')`

5 Implementation

We used **pyDistAlgo** for our implementation of Chord. Our application architecture consists of 3 classes viz., Main, Client and Chord. When the code execution begins, first the setup process takes place wherein the URL from our record set of format : (url A-record/MX-record/NS-record) e.g., (google.com 2001:4860:4860::8888) is hashed using SHA3 and stored in a dictionary with key as the hash-val and value as the record set. Subsequently, each finger table was set up such that each Node points to a specific finger table, and each finger table knows the node to which it belongs, per Fig. 1 below:

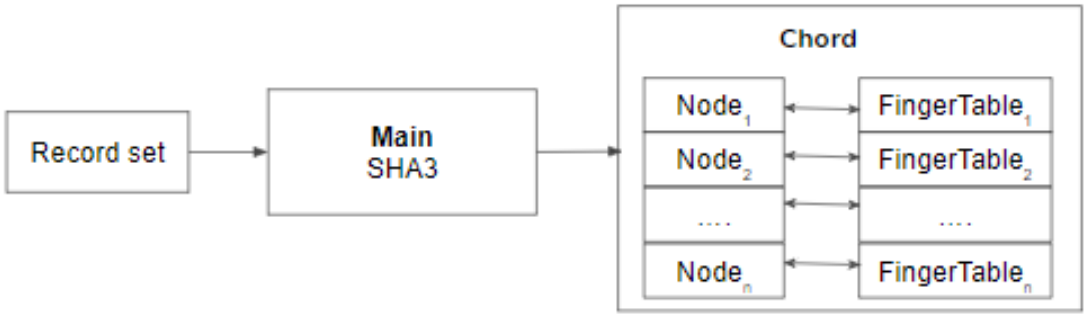


Figure 1: Setup

Following the setup stage, through the CLI interface the user is prompted for a URL address and the record type (A/MX/NS) they want to query, similar to `dig`. It is important that the URL entered is present in our record set so that the code will be able to return a result for the query to the user, hence our code checks if the user record is present in our hash table. If it is present, we relay the URL hash-val to the client class. The client class does the interaction with the Chord class which is actual implementation of the nodes. The Client class keeps pinging the Chord class with the message `find_next_node`. The Chord class keeps relaying the request through the nodes setup starting from the first node. If the Chord class is able to find the hash val in a node, it sends `next_node` message to Client class, subsequently the client class responds with the `get` message to Chord class. The Chord class sends the final `result` message with the response to the user query which is the required record type as requested by user, shown in the following Fig. 2:

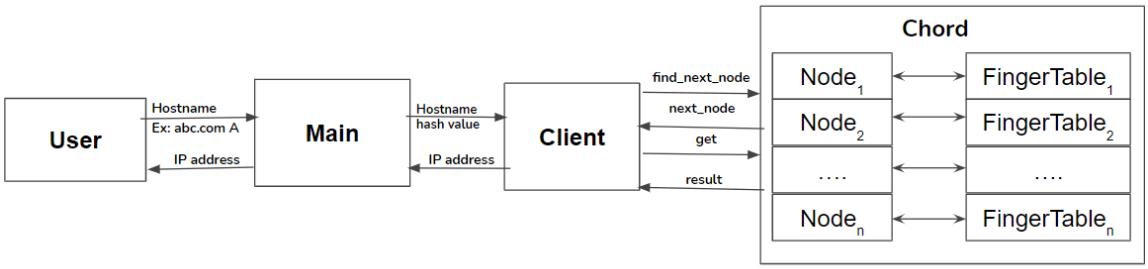


Figure 2: Resolution

6 Experiments

6.1 `dig` versus Chord: UDP

Linux's `dig` tool works primarily over UDP, and DistAlgo's communication channels run over UDP when not given any specific settings ([5],[6]). We obtained the time (in seconds) it took for each of the 400 thousand hostnames to be resolved for each record (MX, NS, A) via `dig`, shown below. Naturally, more popular websites had a shorter query time, e.g. 0.2 seconds, while not-as-popular sites had query times up to 6 seconds. Our results section will break down and discuss the results from general queries in `dig` versus our Chord implementation.

6.2 Node Count versus Lookup time

Here, we analyze lookup time of Chord queries given a set number of nodes. Specifically, we tested with 25, 100, 500, and 1000 nodes through our DistAlgo implementation. We test this on ten thousand (10,000) hostnames and analyze and will later break down our results for both ten thousand and one thousand hostnames.

6.3 Hashing Function versus Lookup time

In this experiment, we analyze different hash functions, SHA-1, SHA-256, SHA-512, and SHA-224, and compare their effects on lookup time. Again, we run these tests on ten thousand hostnames, obtaining results for all four hashing functions. In our results section, we will break down and discuss these results.

6.4 Finger Table Size versus Lookup time

As discussed, the finger table is a Chord node's "routing table" to direct queries between different hops in its resolution. We compare the difference in finger table sizes for small (20), medium (32), and large (64) finger table sizes. Results will be broken down and discussed similarly to the previous sections.

7 Results

7.1 Chord versus DNS lookup times

Our first experiment was comparing raw query times between Chord and `dig`. Our results in Fig. 3 display the difference in results both at a "macro" view of using ten thousand records for a sampling as well as a "micro" view, where we zoom in to one thousand records:

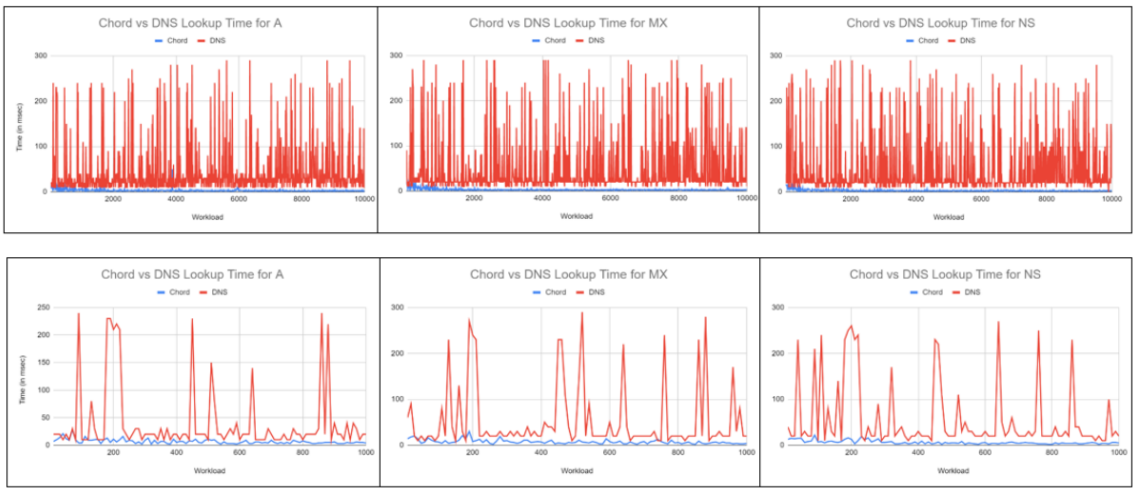


Figure 3: Chord versus DNS

Here, we can see that the regular DNS lookup times (red) take several magnitudes of time longer at most, and even at the smaller cases, Chord (in blue) still holds out with query lookup times.

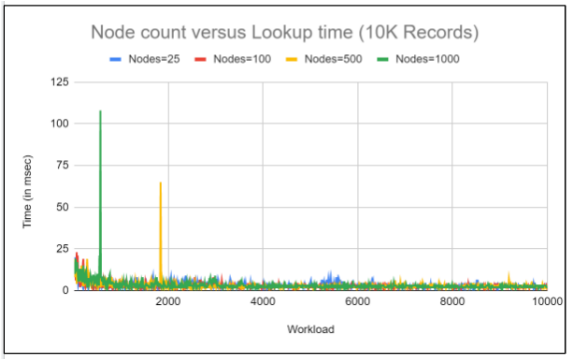
7.2 Node Count versus Lookup times

Surprisingly, there is very little variance in query lookup times with increasing values of nodes in our Chord ring, per Fig. 4a (10K records).

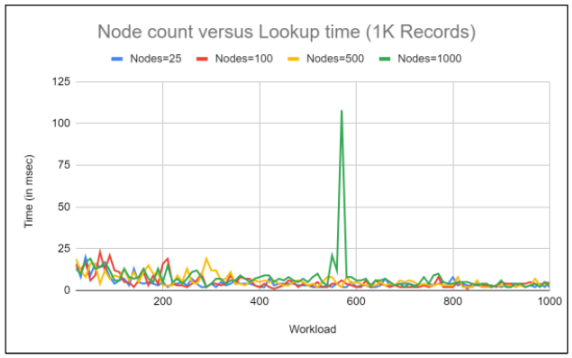
In the graphs shown in Fig. 4, there are two spikes in query times. Much like how real distributed systems work (a la a supercomputer when running a multi-process application), this can be attributed to some random, undefined behavior for the 1000 and 50 node runs. In the event these tests were re-run, there is a strong chance that this would not happen again, or that this may occur in a different spot during the execution.

7.3 Hashing Function versus Lookup time

This experiment compared different hash functions and hashing suites on Chord. SHA-1 (blue), the original hashing function used in Chord, performes substantially slower than SHA512 (red), SHA256 (yellow), and SHA224 (green). The latter three have small variances in lookup time, per the Fig. 5, but they all have much improved, and comparably stable, performance than SHA1.

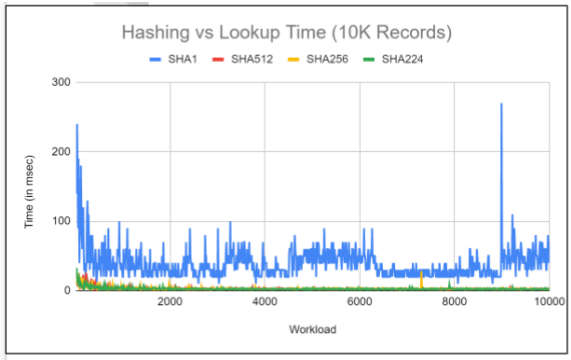


(a) 10K Records

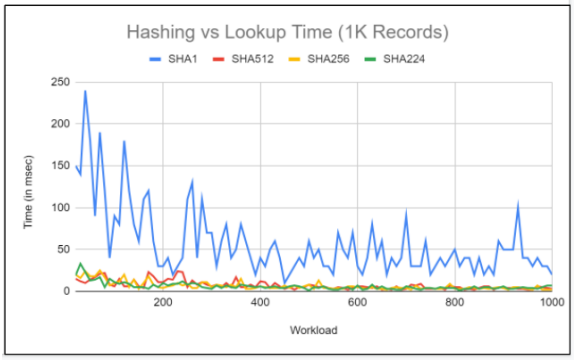


(b) 1K Records

Figure 4: Node Count vs Lookup Time



(a) 10K Records



(b) 1K Records

Figure 5: Hash Function vs Lookup Time

7.4 Finger Table Size versus Lookup time

Different finger table sizes do affect the lookup time of records in the Chord Ring, as shown in Fig. 6:

A large finger table size (64 entries) starts out with the largest variance, as shown on the far left of the graph, though it also has the least fluctuation in lookup time as the workload approaches ten thousand records. Similar trends appear for the small (20) and medium (32) lookup table sizes, though their fluctuations fail to flatten out as much as the large finger table size.

8 Conclusion

In this project, we have shown that our Peer-to-Peer DNS based on Chord Protocol performs 13-17% faster than the existing hierarchical DNS system. It has a high probability of finding the next node hop and resolving a query easily due to its $O(log(n))$ lookup time. With the latest SHA-3

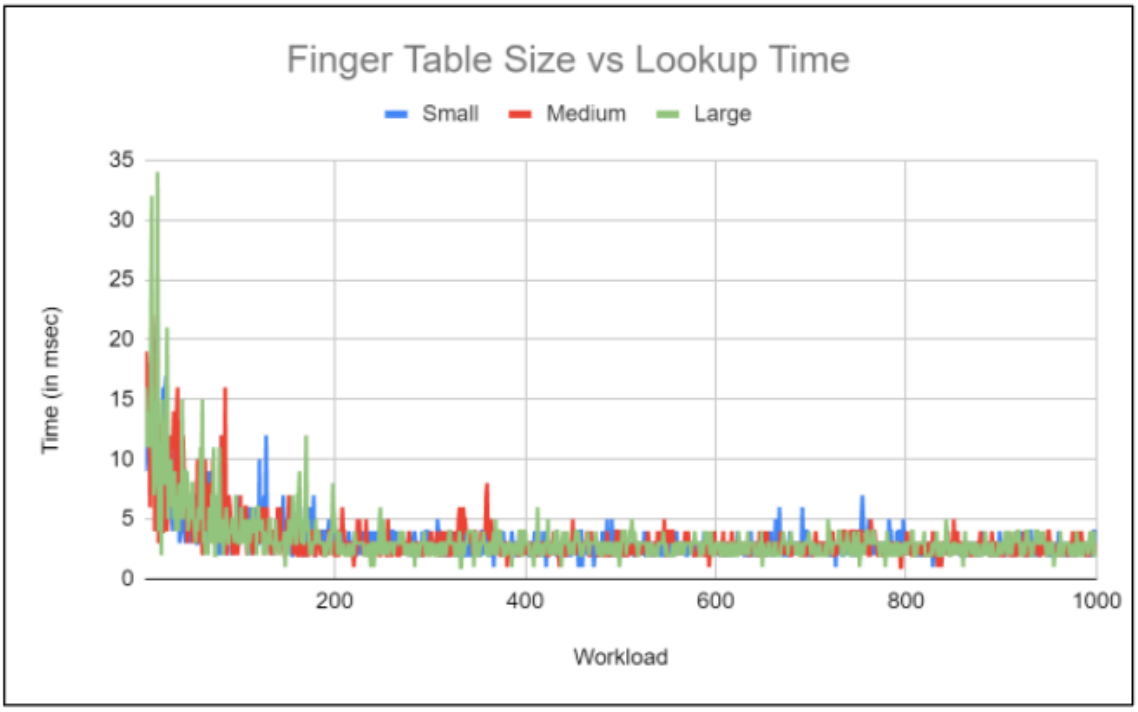


Figure 6: Finger Table Size vs Lookup Time

hashing library, we can also show that the Chord system can already be improved for more secure hashing with less chances of a collision.

9 Future Work

In the original paper, it is stated that a malicious or buggy set of Chord participants could present an incorrect view of the Chord ring. A validation of nodes could be useful for another level of reliable data transfer.

Our results only show what happens with one A, MX, and NS record for DNS queries. Given times and resources, we could scale this to allow multiple instances of each record for one hostname, allowing for increasingly realistic querying.

The original Chord paper deals with addition and deletion of nodes and a stabilization protocol that allows finger table and successor node list adjustments to accommodate the newly added or deleted nodes. This can be a future implementation of our project.

References

[1] Ian Stoica et. al, *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*. In *IEEE/ACM Transactions on Networking* (Volume 11, Issue 1, Feb. 2003, pages 17 - 32)

[2] Russ Cox et. al, *Service DNS using a Peer-to-Peer Lookup Service* 2001, http://pdosnew.csail.mit.edu/papers/chord:dns02/chord:dns02.pdf.

[3] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris, *DNS performance and the effectiveness of caching*. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop '01, San Francisco, California*, November 2001. http://nms.lcs.mit.edu/papers/dns-ton2002.pdf

[4] Mozilla's Cipherscan Repository: "Alexa Top 1 million Websites". https://github.com/mozilla/cipherscan/tree/master/top1m

[5] Yanhong A. Liu et. al, *From Clarity to Efficiency for Distributed Algorithms*. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 395-410. https://www3.cs.stonybrook.edu/ liu/papers/DistPL-OOPSLA12.pdf

[6] Yanhong A. Liu et. al, *DistAlgo Language Description*, revised as of 2017. https://www3.cs.stonybrook.edu/ liu/distalgo/language.pdf

[7] Ian Clarke et. al, *Freenet: A Distributed Anonymous Information Storage and Retrieval System*, in *International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*; http://www.cs.cornell.edu/people/egs/615/freenet.pdf

[8] A. Bakker et. al, *The Globe Distribution Network*. In *Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference*; https://www.researchgate.net/publication/220881505_The_Globe_Distribution_Network