Project report : DNS on Chord

https://github.com/soumyadeep2007/dns_chord

Anupam Samanta, Jatin Garg, Soumyadeep Chakraborty

May 2018

Grading detail

For 30 percent grade, please refer to sections [4], [5] and [6].

1 Abstract

A peer to peer DNS service eliminates the coupling between service and administration that is inherent in traditional hierarchical DNS. An overwhelming source of incorrect replies or DNS query failures is borne out of name-server administrative errors[1, 2]. Implementing such a service on top of Chord [3], a distributed hash table, not only remedies this problem, but also confers benefits such as fault-tolerance through replication and load-balance. [4] explores this idea at considerable depth and we look towards it as our main source of inspiration. Studies[2] have shown that as much as 18% of overall DNS traffic is towards the root servers. A peer-to-peer system essentially eliminates any hierarchy whatsoever, and by extension, obviates the need to have root or TLD servers, providing for better load balance. This project, through an implementation of DNS on top of Chord, has provided an opportunity to study the aforementioned aspects. The implementation is done on top of a research language, DistAlgo [5], which provides powerful primitives for distributed programming. A comparative performance study of our DNS service with traditional DNS is performed. For our experiments, we utilized a sizeable dataset of AAAA records which we obtained from the Forward DNS dataset [6].

2 Introduction

2.1 P2P DNS

A P2P DNS system, such as the one we propose is different from a traditional DNS in these ways:

1. There is no administrative hierarchy coupled to the ownership of records. Any node can house any record. Records from the same domain may map to entirely different nodes. Records can be added/removed freely without having the need to contact any administrative body.

2. Since there is no hierarchy in the system, certain types of DNS records do not have any value anymore. For example: the NS record is extraneous.

3. A P2P DNS can work with both recursive and iterative query resolution. The advantages of a recursive query scheme in a P2P DNS outweighs the advantages if a similar scheme is adopted in traditional DNS. In traditional DNS, a recursive query scheme would burden the servers at the top of the hierarchy, as they receive maximum traffic. P2P DNSes have no similar hotspots, as they don't have hierarchy. Thus a DNS server can readily forward requests to another server in a P2P scheme.

2.2 Chord

Chord is a distributed lookup protocol, implemented as a distributed hash table, which addresses a common concern of P2P systems : efficient location of a node storing an item. The efficiency of Chord is inherent in its guarantee to answer any query in O($log_2 N$) messages where N is the number of nodes in the system. The beauty of Chord lies in the fact that this guarantee is achieved regardless of which node receives the initial query. Chord provides a simple high-level API for:

1. Mapping a key-value pair to a node.

2. Locating the node to which a particular key is mapped.

3. Querying the node after locating it to obtain the desired key-value pair.

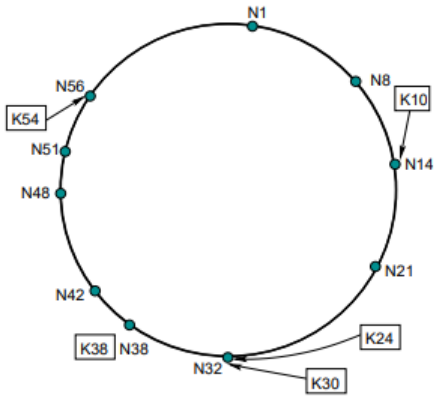


Figure 1: Chord ring having 10 nodes storing 5 keys

2.2.1 Consistent Hashing

Chord relies on Consistent Hashing[7] to assign keys to nodes. In this scheme, both keys for data and for nodes are hashed to different positions in an identifier circle, with keys ranging from 0 to $2^m - 1$. The circle represents a key space of 2^m keys. Each position in the ring is represented as an m-bit identifier. Please note that throughout this report, we have referred to keys and identifiers interchangeably. Identifiers are obtained as follows:

1. Identifiers for data are obtained by hashing their keys with SHA-1[8].
2. Identifiers for nodes are obtained by hashing their IP addresses with SHA-1.

To avoid collisions in the assignment of identifiers, techniques such as Universal Hashing can be adopted.

Consistent hashing works in the following manner: Identifiers are ordered in the circle modulo 2^m. Key k is assigned to the first node whose identifier either is equal to k or succeeds k in the identifier circle in the clockwise direction. Such a node is defined as *successor(k)*. Similarly, *predecessor(k)* can be defined. As depicted in Fig. 1, the successor of K10 is N14 since 14 follows 10 in the Chord ring. In general, for a node with identifier i will hold keys lying in the interval: (predecessor(k), i].

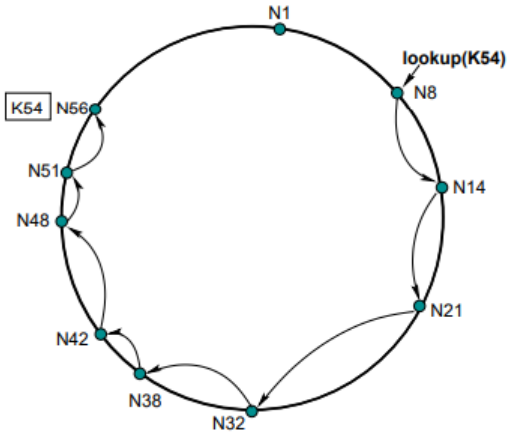


Figure 2: Simple key location: Message sequence for key 54.

2.2.2 Key Location

Depending on the amount of state maintained at each node about its successors, the number of messages that need to be sent to locate a key will scale.

Simple Key Location is a scheme in which each node only stores the network address of its immediate successor. In the worst case, lookup would take O(N) messages if the Chord ring has N nodes. The sequence of messages is depicted for a lookup on the key 54 in Fig 2.

Scalable Key Location is a scheme in which each node stores the network addresses of $O(log_2 m) \approx O(log_2 N)$ successors, in a data structure called a *finger table*. The ith entry in this table at node n will house s = successor(n + 2^{i-1}). Now lookup can be done with a binary search on the table. To sum up, in the worst case this scheme would incur an overhead of $O(log_2 N)$ messages. The finger table for the initial node 8 and message sequence for resolving the key 54 starting at node 8 is depicted in Fig 3.

2.3 DistAlgo

The language supports distributed programming at a very high level with high level constructs for: (1) distributed processes, (2) sending and receiving messages, (3) synchronization conditions expressed as high-level queries over message history sequences, (4) process and communication configuration.

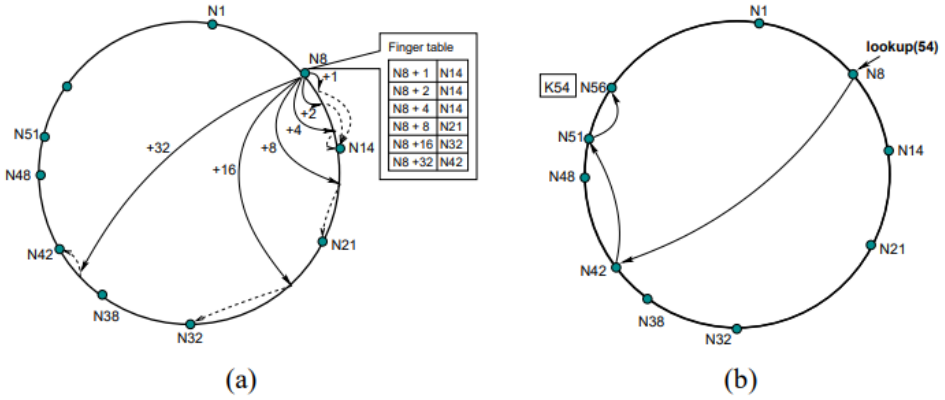


Figure 3: Scalable key location: a) Finger table at node 8. b) Message sequence for key 54.

One of the key constructs in the language are the *send* and *receive*. Specific tuples are sent out to specified nodes. Receivers are set up to receive messages with specific tuple patterns.

3 Approach

3.1 Record format

We only consider DNS AAAA records (which is a hostname and IPv6 address pair) for purposes of experimentation, given the number of AAAA records in our dataset was apt. Our experiments could be conducted just as easily with other DNS record as well.

Format : ('hostname', 'IPv6 address')

3.2 DistAlgo setup

We have one DistAlgo process, the *main* process, setup the Chord *nodes* as separate processes and a *resolver* process. All of these processes execute on the same physical machine, but can be made to execute across physical machines via DistAlgo's multi-host configuration. As a key part of its initialization, each Chord node receives the key-value pairs that it will house along

with its finger table entries. Finger table entries are calculated at the main process after all nodes have been created, and sent to the nodes upon their initialization. The resolver, during setup, is fed its workload: a list of DNS AAAA queries that it has to resolve. We ran DistAlgo over an unreliable UDP channel.

3.3 Name Resolution

Each process communicates with each other via messages. Fig. 4 depicts a typical workflow to resolve the key 'kx' which involves the following messages:

1. The resolver sends a 'find_successor' message to a node picked at random. In the example this node is 'i'.

2. The finger table at 'i' is looked up and it is determined that the next closest node to 'kx' is 'm'. So a 'find_successor' message is sent to 'm' from 'i'.

3. The finger table at 'm' is looked up and it is determined that the next closest node to 'kx' is 'p'. So a 'find_successor' message is sent to 'p' from 'm'.

4. The finger table at 'p' is looked up and it is determined that the successor for 'kx' is 's'. This information is sent across to the resolver from 'p' in a 'successor' message.

5. Having received the 'successor' message the resolver now has located the node('s') where 'kx' resides. It now sends across a 'get' message to 's' for 'kx'.

6. The node 's' responds with a 'result' message to the resolver which contains the key-value pair: 'kx': 'kxv'. Please note: if the key 'kx' was not part of the ring (i.e. we looked up a DNS name that does not exist), a null value is returned by 's' as part of a 'result' message.

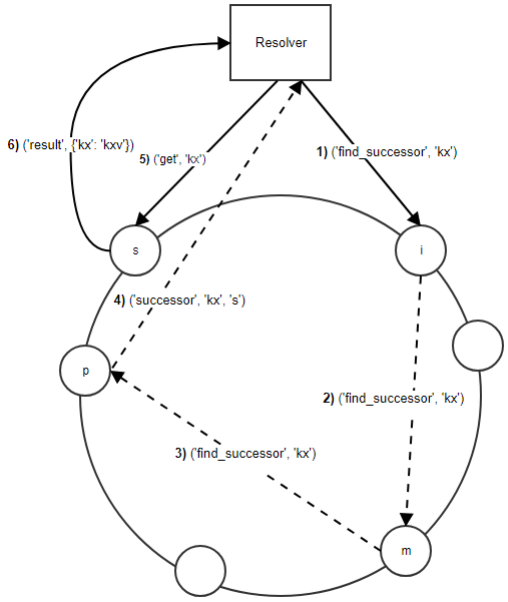


Figure 4: A typical workflow of messages for resolving the key 'kx'

4 Evaluation

4.1 Experimental setup

DistAlgo can be executed over both reliable (TCP) and unreliable (UDP) channels via a configuration parameter. We chose to do both and we have compared their performance.

The Forward DNS dataset was easy to use and we applied it both on our Chord implementation and traditional DNS with the help of an online Dig tool[9]. We conducted the experiments with a random subset of these records. We randomly sampled 10,000 AAAA DNS records from the dataset. These 10,000 records have 8328 2nd level domains, following a zipfian distribution.

4.2 Performance Study

For the performance study, we collected the following metrics: (1) average query resolution time (2) average number of DNS servers (hops) involved in

resolving a query (3) average network latency and (4) node load distribution.

The online dig tool was configured to bypass the local DNS cache every time it resolved a query. By enabling the trace option, we could collect the hops metric as well as measure the average network latency between the dig tool and each remote DNS server, authoritative or otherwise to arrive at the average latency metric.

From our Chord implementation, we could easily derive the hops metric, simply incrementing a counter as a query request travelled across nodes. We had to simulate network latency in our implementation in order to enable a fair comparison. This was because the entire Chord ring was executed on a single physical machine. To simulate this, we inserted a sleep whenever any process received a message from another process. The sleep duration was randomly sampled from the Gaussian distribution of network latencies as gathered from the dig tool.

4.3 Parameters

The parameters that we varied for our Chord implementation are:

1. m
2. N - the number of servers

4.4 Results

4.4.1 Number of hops

	DNS	Chord
Mean	3.86	4.30
Median	4.00	4.00
Std. dev.	0.54	1.04
Min	1.00	2.00
Max	5.00	7.00

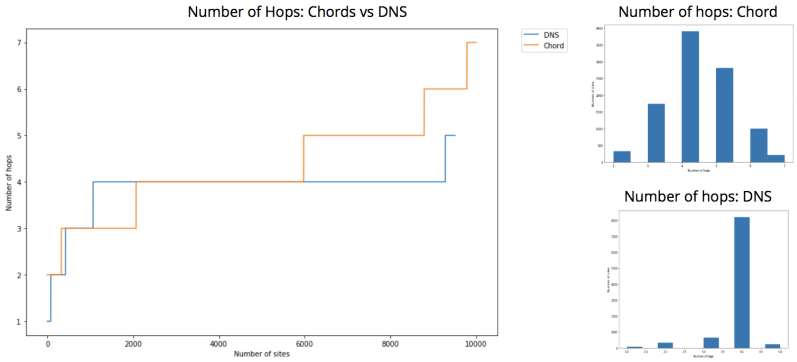


Figure 5: Comparative study of number of hops for N=30.

We define the number of hops with respect to DNS resolution as the number of DNS servers that are involved in serving a DNS query. We compared the number of hops metric for our Chord implementation with that of DNS.

Fig 5 shows the number of hops comparison for a N=20, m=20 Chord ring versus traditional DNS. We can see that traditional DNS outperforms Chord. While traditional DNS requires 3.86 hops as compared to 4.30 hops for Chord on average.

If we decrease N, the number of hops for our Chord implementation decreases, as expected. This is depicted in Fig 6. This might indicate that decreasing N is a great way to reduce the number of hops, and in turn, reduce latency for query resolution. However, there lies the caveat that decreasing the number of servers would increase the load on each server. We have not considered the number of DNS records as an experiment parameter. We have assumed that the cost of looking up a DNS record is negligible as compared to network latency between two nodes. For our experiments with 10,000 records, such an assumption is viable. However, for a real world setting, there would be many more records and processing time must be considered. Not only would processing time at each node increase by reducing N, fault tolerance would also suffer as number of replicas for each record would also

have to decrease with N. Reducing N would also go against the philosophy of our system, where we are using commodity servers to serve DNS queries. To handle larger loads, commodity servers would not be enough.

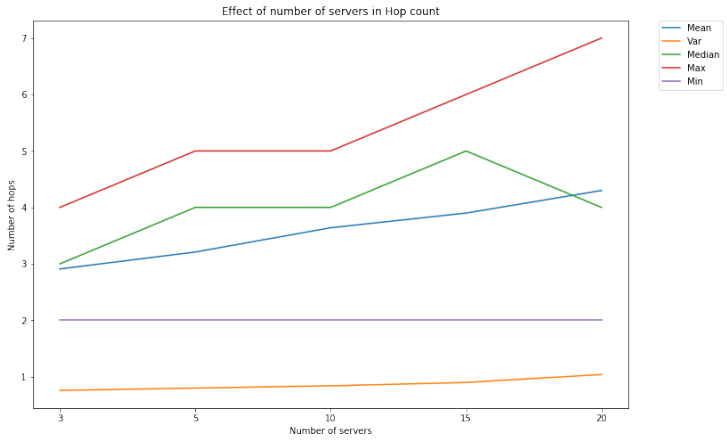
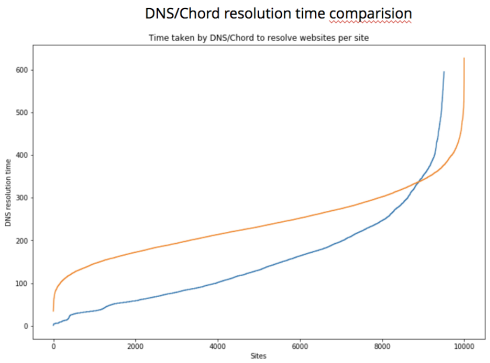
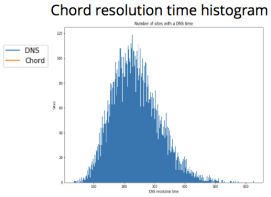


Figure 6: Effect of N on hop count. Varying m did not make much of an impact on number of hops. We believe that this has to do with the number of DNS records that we used. We would have to had more than 10,000 records to see an impact after varying m.

4.4.2 DNS resolution latency

	DNS	Chord
Mean	147 ms	239 ms
Median	124 ms	232 ms
Std. dev.	104 ms	77 ms
Min	1 ms	34 ms
Max	595 ms	627 ms





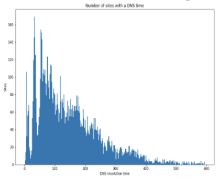


Figure 7: Comparative study of resolution time for N=30.

We compared the average resolution latency of our Chord implementation against that of traditional DNS. We modelled real world latency by using a kernel density estimator on a distribution of network latencies obtained from making traditional DNS requests. We sampled from the kernel distribution to inject latency on every distinct network call.

The results were not surprising considering our results of the number of hops metric. Since, the processing time at each node is negligible given our small number of DNS records, average resolution latency is directly correlated with number of hops. Trivially, an increase in the number of hops increases the number of network calls, and in turn, increases the resolution time. The results are shown in Figure 7, for m=20, N=30.

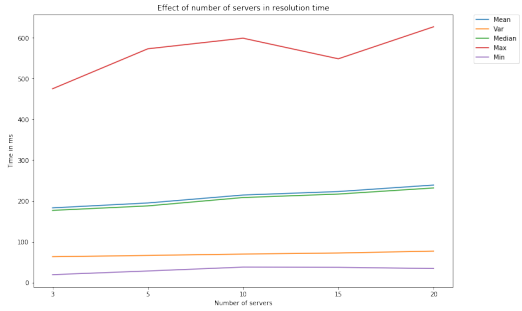


Figure 8: Tweaking N to see effect of N on resolution time. Varying N yields a similar curve as the number of hops, as is expected. Varying m did not make any impact for reasons similar as above.

4.4.3 Node load distribution

Figure 9 shows the number of requests that a certain node participated in resolving either yielding an intermediate reply or an authoritative response, for m=20, N=30. As we can see in the figure, the load is distributed quite randomly. This is in stark contrast to traditional DNS where the nodes near the root of the hierarchy are involved in processing each and every query. This is consistent with what we had set to achieve.

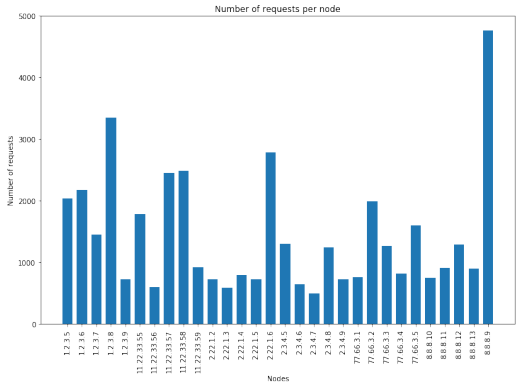


Figure 9: Alternative visualization for node load.

As we increase N, we find that the load per server decreases exponentially, as expected. We can explain this as follows: The expected number of queries that a node would participate in (E) = (Average no of hops) * (Number of requests) / (Number of servers). We can see that as the hop count increases slowly as we increase N and given our fixed number of requests, the node load falls rapidly initially but shows a lesser rate of decrease at higher values of N. Varying m did not make any impact for reasons similar as above.

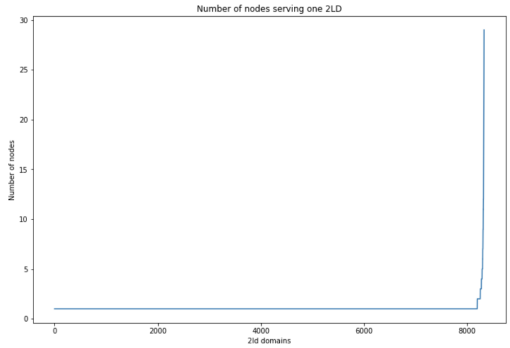


Figure 10: Alternative visualization for node load.

In Figure 10 x-axis depicts 2LDs sorted on their counts in the dataset. For example: the 8305th value on the x-axis represents a 2LD: google.com which has 55 URLs. The y-axis gives the number of nodes which contains these URLs. We can see that the higher the count of URLs under a TLD,

greater is the number of nodes serving it, thereby achieving a more uniform distribution.

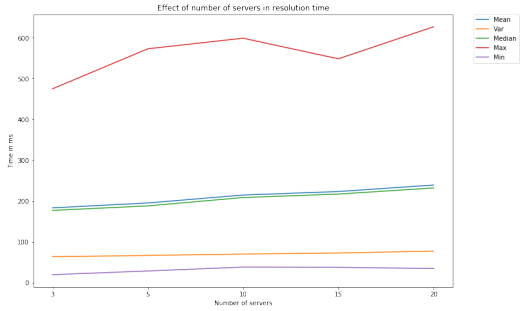


Figure 11: Effect of N on node load distribution. Varying N has no effect on the load balance. Varying m did not make any impact for reasons similar as above.

5 Discussion and Future Work

We found that the number of servers involved in resolving a DNS query in traditional DNS was far less as compared to our Chord implementation. This can be attributed to the large branching factor at the root of the DNS hierarchy that is found in traditional DNS. In our system, we cannot do better than O($log_2 N$) messages in a Chord ring having N nodes. If we implemented Pastry or Kademlia which are improvements over Chord, we could have guaranteed message latencies around O(log_{16}N). Our work can be extended to:

1. Implement replication of key-value pairs using Chord successor lists, a feature which replicates key-value pairs belonging to a node in n successor nodes. This feature is a large improvement over traditional DNS where replication is optional throughout the hierarchy and depends on the resources available to the concerned institution.

2. Implement caching and indexing of DNS records for faster retrieval, a feature that is consistent in traditional DNS.

3. Implement Chord's node join and stabilization protocol with which nodes can be added or removed from the system dynamically. Upon such events, keys would need to be dynamically reassigned to servers.

6 Conclusion

We first implemented Chord, a distributed hash table which serves as a generic P2P lookup service. Then we implemented a DNS on top of Chord, with support for any DNS record type.

Such a system has advantages such as:

1. Better load balance: This is evident by our experiments where we calculated how many requests each node served. The load distribution was quite uniform across the nodes irrespective of number of servers in system. This is an advantage over traditional DNS where the root and TLD servers have to serve almost every other request.

2. Average resolution latencies: In Chord, as there is no hierarchy, the hops traversed in order to find authoritative name server can be more than the traditional DNS server. However the good thing is that DNS queries can sometimes be resolved in one hop which is not possible in case of traditional DNS.

3. Proofing from DDoS attacks: Chord is more reliable in preventing DDos attacks as there is no single server serving an entire domain. So we cannot bring them down all at once by taking out one server, which is quite contrary to traditional DNS.

4. Elimination of painful DNS server administration: This is also eliminated in Chord as there is no administrative hierarchy.

5. Commodity servers: Chord can be implemented with commodity hardware as compared to the super-servers required towards the root of a traditional DNS hierarchy.

With all the features that we suggested in future work, this system of resolution of DNS queries through Chord could prove extremely useful.

7 References

1. P. Danzig, K. Obraczka, and A. Kumar. An analysis of wide-area name server traffic: A study of the internet domain name system. In Proc ACM SIGCOMM, pages 281–292, Baltimore, MD, August 1992.

2. 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.

3. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proc. ACM SIGCOMM, San Diego, 2001

4. Russ Cox, Athicha Muthitacharoen, Robert T. Morris. Serving DNS using a Peer-to-Peer Lookup Service

5. Yanhong A. Liu, Scott D. Stoller, Bo Lin, Michael Gorbovitski. From Clarity to Efficiency for Distributed Algorithms

6. Forward DNS dataset. `https://github.com/rapid7/sonar/wiki/Forward-DNS`

7. Consistent Hashing KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing (El Paso, TX, May 1997), pp. 654–663.

8. SHA-1. FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

9. Online dig tool. `https://www.digwebinterface.com/`

10. DNS_Chord repository `https://github.com/soumyadeep2007/dns_chord`