SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.041 Distributed Systems and Computing

DNS on Chord

Team 1

Rohit Raghuram Murarishetti 1005398

Sanat Khandekar 1005281

Mohammed Fauzaan Mahaboob 1005404

Harikrishnan Chalapathy Anirudh 1005501

Han Wei Guang 1005233

DNS on Chord

Mohammed Fauzaan, Sanat Khandekar, Rohit Raghuram Murarishetti, Harikrishnan Chalapathy Anirudh, Han Wei Guang

Supervisor: Professor Sudipta Chattopadhyay

Abstract:

To access the GitHub repository for this project, kindly refer to this link. Additionally, for comprehensive open-source documentation on Go-lang pkgsite, please visit this link.

One of the essential features of the modern Internet is the domain name system (DNS), which is the naming system that translates human-readable domain names to their corresponding Internal Protocol(IP) addresses [4].

However, traditional DNS architecture may have various performance inefficiencies and vulnerabilities regarding security and outages. DNS implements a hierarchical structure, which can serve requests from clients either recursively or iteratively. The lookup times associated with such a system tend to be variable depending on potential caching, and the amount of load each level needs to take on may also be unequal. As a result, the TLDs and upper-level servers can become a single point of failure, making it susceptible to DDoS attacks. Based on the limitations of the current version of DNS, we came up with the following problem statement:

How might we modify and improve on the present-day protocol of DNS to address its current weakness?

Our proposed solution aims to achieve the following:

1. Resilience to a single point of failure.

2. Seamless node joins and departures

3. Correct, fast and scalable DNS lookups.

4. Improved load balancing through distributed nodes.

Contents

Chapter 1

The Chord Protocol

Chord is a distributed lookup protocol, that stores data as key-value pairs at the nodes of the network, using consistent hashing to assign keys to the nodes, thus providing a degree of load balancing [2]. Chord also adapts efficiently as nodes join and leave the system, and guarantees an efficient location of data items, making it a promising alternative to existing DNS architecture.

A distributed indexing system based on Chord thus aims to mitigate the issue of load balancing in Traditional DNS architecture through nodes in a distributed system that can effectively look up domains without having to query vertically between different levels.

Firstly, Chord assigns keys to nodes through consistent hashing, so each node will have approximately the same number of keys. The key, with its unique identifier defined as k, is assigned the first node whose identifier is equal to or follows the identifier of k. This node is known as the successor node of key k.

Figure 1.1 shows a chord ring, with 10 nodes and 5 keys. The successor of key 9 is node 14, the successor of keys 25 and 32 is node 32, and the successors of keys 48 and 52 are nodes 48 and 56 respectively.

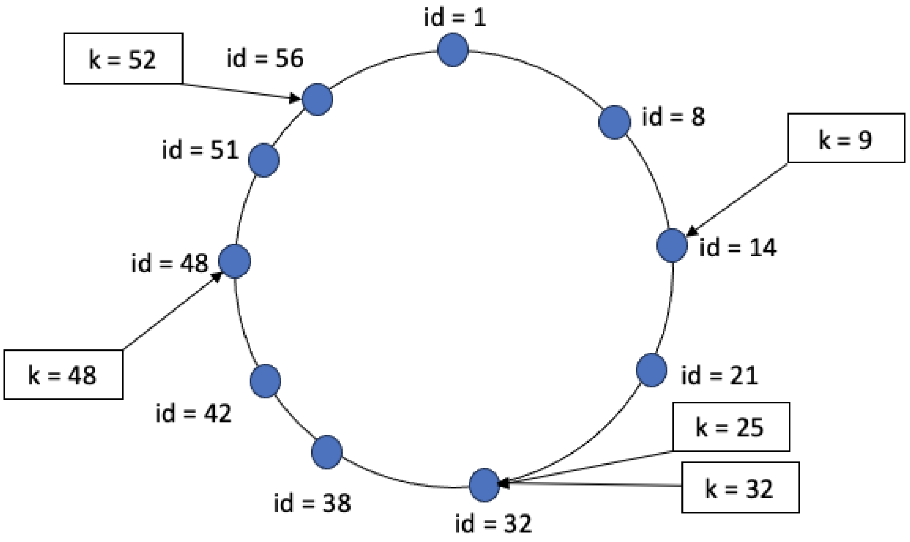


Figure 1.1: A chord ring with 10 rings and 5 keys

With consistent hashing, only $O\left(\frac{1}{N}\right)$ fraction of the keys are redistributed when the n^{th} node leaves or joins the network. When node n joins the network, some keys belonging to n's successor will be reassigned to n. For example, if node 27 joins the network, key 25 from node 32 will be reassigned to it, as shown in Figure 1.2.

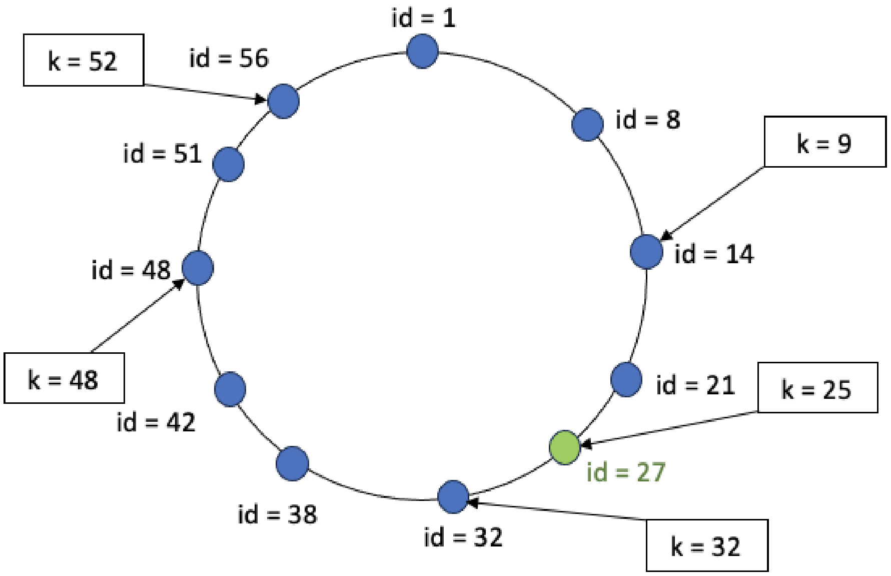


Figure 1.2: Key reassignment during node join

Conversely, when n leaves the network, all of its keys will be reassigned to n's successor. This is illustrated in Figure 1.3, where node 32 leaves and all its keys are reassigned to its successor, node 38.

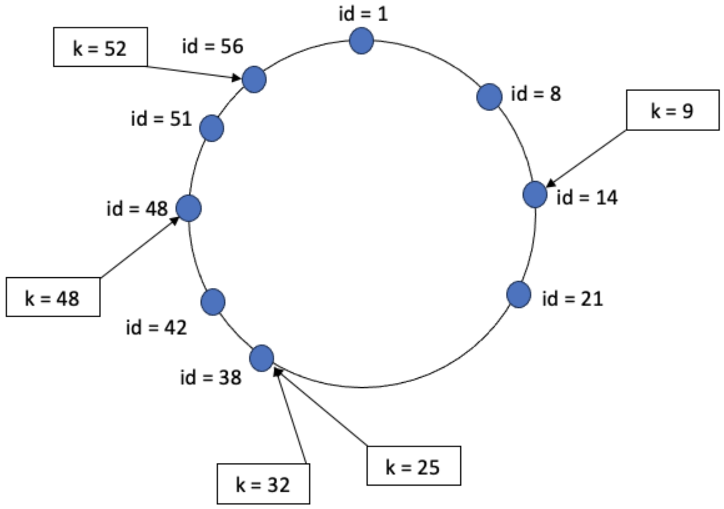
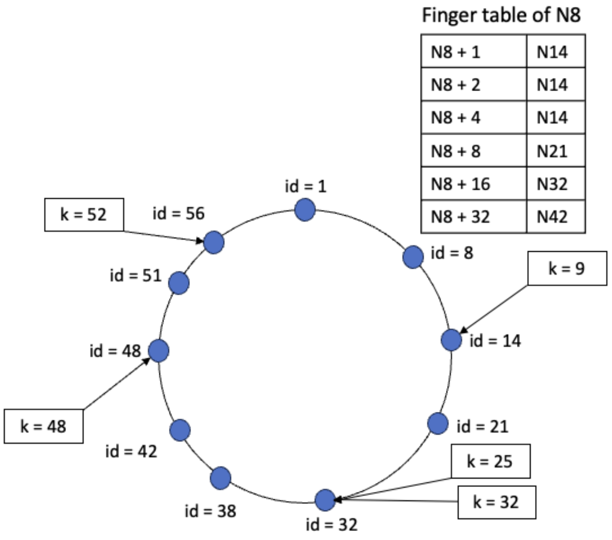


Figure 1.3: Key reassignment during node departure

Furthermore, each node also maintains a routing table, known as a finger table. The i^{th} entry in the table at node n contains the identity of the first node that succeeds n by at least 2^{i-1}, where $1 \leq i \leq m$, and m is the identifier length. One such example of a node's finger table is shown in Figure 1.4, where the finger table of node 8 is illustrated.


Finger table of N8

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 + 16	N32
N8 + 32	N42

Figure 1.4: Finger table of node 8

Chapter 2

System Overview

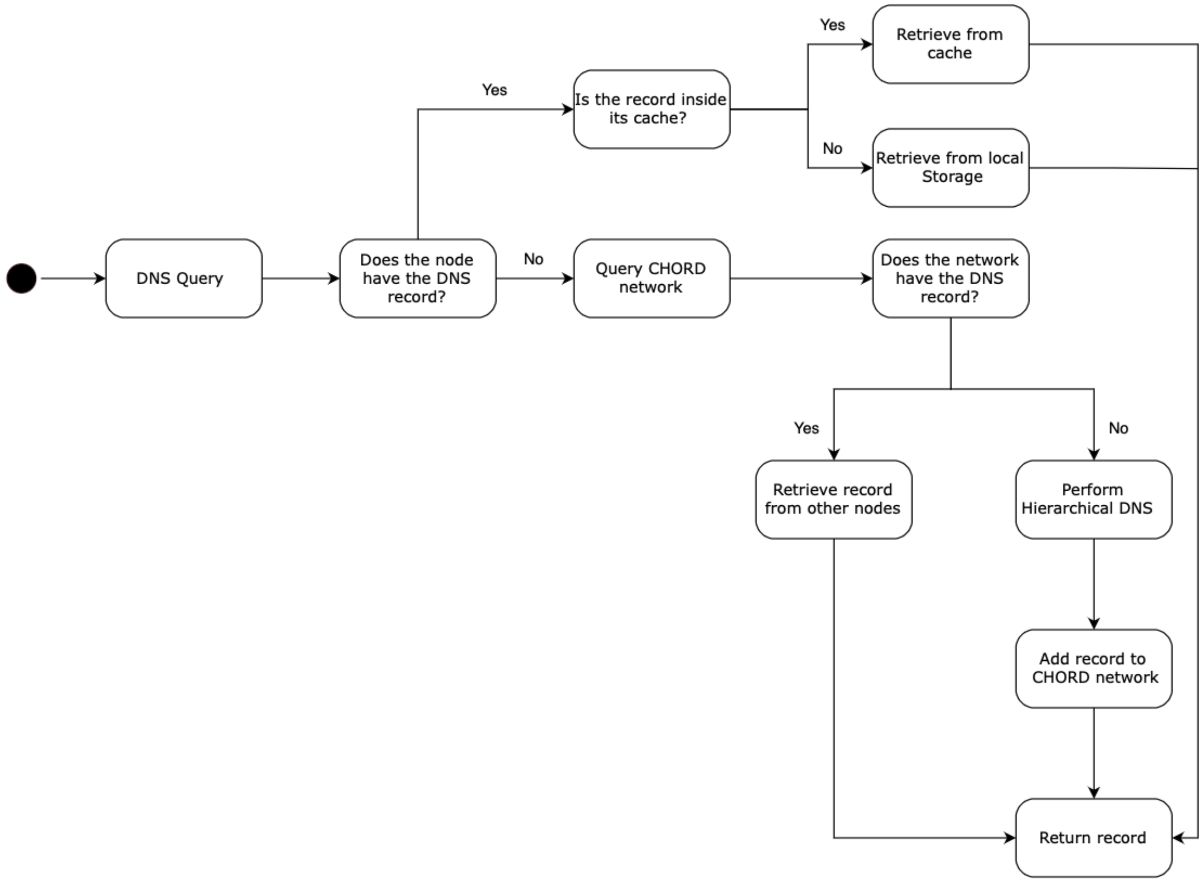

DNS Query
Does the node have the DNS record?
Yes
No
Is the record inside its cache?
Yes
Retrieve from cache
No
Retrieve from local Storage
Query CHORD network
Does the network have the DNS record?
Yes
No
Retrieve record from other nodes
Perform Hierarchical DNS
Add record to CHORD network
Return record

Figure 2.1: Flow chart diagram of a DNS query

Our DNS system builds on top of the Chord protocol, consisting of multiple nodes, where each node stores some records, either in its storage or local cache. The diagram above, Figure 2.1, summarises what happens in our system when a user performs a DNS query.

If the node being queried contains the DNS records, it will retrieve it, either from the local storage or from the local cache, if the same DNS query has been performed before. If the node does not contain the DNS record, we will find the node that holds this requested DNS record, should it exist in the network. If the requested DNS record exists, that particular node will then return the requested DNS record.

If the DNS record is not present in our network, we will perform a traditional DNS query to obtain and return the record, and insert the record into our network for future lookups.

Chapter 3

Design

3.1 Features

At its core, our project delivers the following features of the chord protocol:

- **Porting over of DNS records from DNS servers to a chord network**.
- **Correctness**: Lookup of domains that return the correct domain-IP mapping using any node in a distributed system.
- **Availability and Fault Tolerance**: Account for failed nodes or new nodes in the system.
- **Scalability**: It should efficiently provide mapping, given a large number of domain-IP pairs in the hash table.
- **Decentralisation**: Fully distributed with no one node more important than the other.

3.2 Porting over of DNS records from DNS servers to a chord network

A crucial aspect of our project involves migrating DNS records from traditional DNS servers to the Chord network. It is important to note here that the scale of implementation proposed in this project refers to chord networks that exist within subnets, and we do not seek to replace the entire DNS system with a chord P2P invariant. The proposed implementation explores if peering systems within subnets can improve the performance of DNS through mechanisms like caching, and local storage.

According to our system architecture diagram, when a queried node possesses the DNS records associated with the query, it retrieves them either from local storage or from the local cache.

Firstly, we will check if the hashed website is present in the cache. If it is, retrieve the IP addresses from the cache.

```go
ip_addr, ok := node.CachedQuery[hashedWebsite]
if ok {
    log.Info().Msg("Retrieving from LRUCache")
    for _, ip_c := range ip_addr.value {
        log.Info().Msgf("> %s. IN A %s", website, ip_c)
    }
}
```

Listing 3.1: Retrieving from LRUCache

Should the website not be present in the cache, the system proceeds to examine the local storage linked to the current node. If the website is indeed found in local storage, the associated IP addresses are then retrieved.

```go
ip_addr, ok := node.HashIPStorage[node.Nodeid][hashedWebsite]
log.Info().Msgf("> The Website %s has been hashed to %d", website, hashedWebsite)
if ok {
    log.Info().Msg("Retrieving from Local Storage")
    for _, ip_c := range ip_addr {
        log.Info().Msgf("> %s. IN A %s", website, ip_c)
    }
}
```

Listing 3.2: Retrieving from Local Storage

If the website is not in the local storage, we query the Chord network to find the successor node that holds this requested DNS record. We use an RPC call to request information about the website from the identified successor node.

```go
succPointer, hopCount := node.FindSuccessor(hashedWebsite, 0)
log.Info().Msgf("> Number of Hops: %d", hopCount)
// log hopcount into the log file using the library
log.Info().Msgf("> The Website would be stored at its successor Nodeid: %d IP: %s", succPointer.Nodeid, succPointer.IP)
msg := message.RequestMessage{Type: GET, TargetId: hashedWebsite}
reply := node.CallRPC(msg, succPointer.IP)
if reply.QueryResponse != nil {
    log.Info().Msg("Retrieving from Chord Network")
    for _, ip_c := range reply.QueryResponse {
        log.Info().Msgf("> %s. IN A %s", website, ip_c)
    }
}
```

Listing 3.3: Query the Chord network

In the event of no response from the Chord network, our system resorts to a conventional DNS lookup using Go's net package. Employing the LookupIP function, we initiate a DNS query to retrieve the IP addresses linked to the specified website.

```go
ips, err := net.LookupIP(website)
if err != nil {
    log.Error().Err(err).Msg("Could not get IPs")
    return
}
ip_addresses := []string{}
log.Info().Msgf("IP ADDRESSES %v", ip_addresses)

for _, ip := range ips {
    ip_addresses = append(ip_addresses, ip.String())
    log.Info().Msgf("> %s. IN A %s", website, ip.String())
}
```

Listing 3.4: Conventional DNS Lookup

Following a successful DNS lookup,

- We will update the cache with the DNS lookup result.
- Notify the Chord successor node about the new data by initiating an RPC call to the successor node in the Chord network. It sends a message of type PUT along with the target node ID and a payload containing the hashed website as a key and the IP addresses as values.
- After making the RPC call, we check if we receive an acknowledgment from the successor node.
- As part of our optimization strategy, we monitor the size of our cache. If the length of the cache exceeds a predefined size limit, we identify the oldest entry based on the timestamp. We then remove that entry from the cache, ensuring that the cache size is within the specified limit.

```go
node.CachedQuery[hashedWebsite] = LRUCache{value: ip_addresses, cacheTime: node.CacheTime}
reply = node.CallRPC(message.RequestMessage{Type: PUT, TargetId: succPointer.Nodeid, Payload: map[uint64][]string{hashedWebsite: ip_addresses}}, succPointer.IP)

if reply.Type == ACK {
    if len(node.CachedQuery) > CACHE_SIZE {
        var minKey uint64
        minValue := uint64(18446744073709551615)
        for key, value := range node.CachedQuery {
            if value.cacheTime < minValue {
                minKey = key
                minValue = value.cacheTime
            }
        }
        if minKey != 0 {
            delete(node.CachedQuery, minKey)
        }
    }
} else {
    log.Error().Msg("Put failed")
}
```

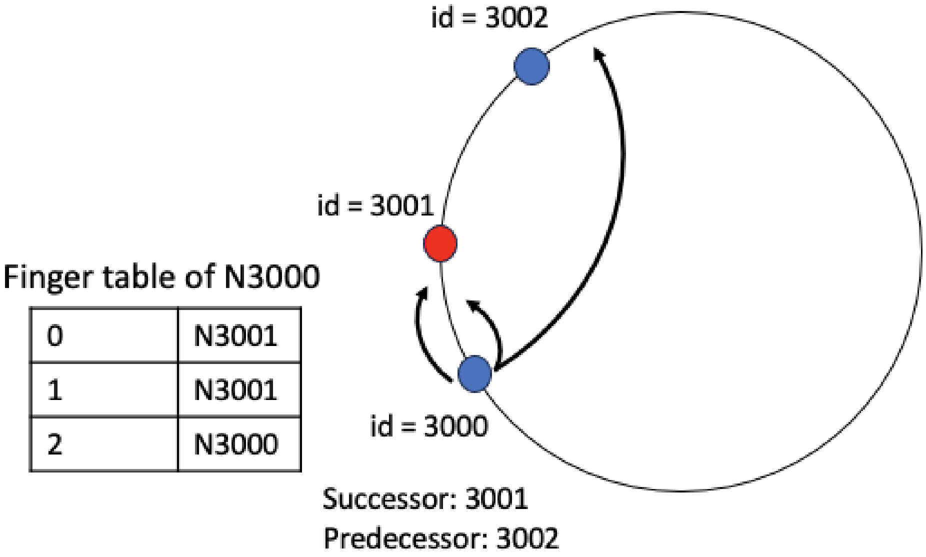
Listing 3.5: Actions taken after a successful DNS Lookup

3.3 Correctness

3.3.1 Ring structure correctness

Zave(2015) highlights that the chord network is not correct as per the original implementation discussed in the paper.[5] As such we faced issues when it came to nodes that hashed in slightly different ways. Anomalies in small networks, such as falling below the minimum ring size or having successor lists that "wrap around" the ring, can lead to correctness problems. This posed an issue in terms of correctness, and the proposed implementation in our project aims to overcome this limitation.

To ensure the correct upkeep of the ring structure in the event of node failures, we implement a Successor List that is based directly on the Pointer data structures rather than relying on its FindSuccessor() method. This is a list of a node's *k* successors. Consider a ring structure that has nodes with values: 3000, 3001, and 3002. Node 3000's Successor List (including itself) would be [3000, 3001, 3002]. To illustrate the incorrectness of not having the Successor List, refer to Figure 3.1:



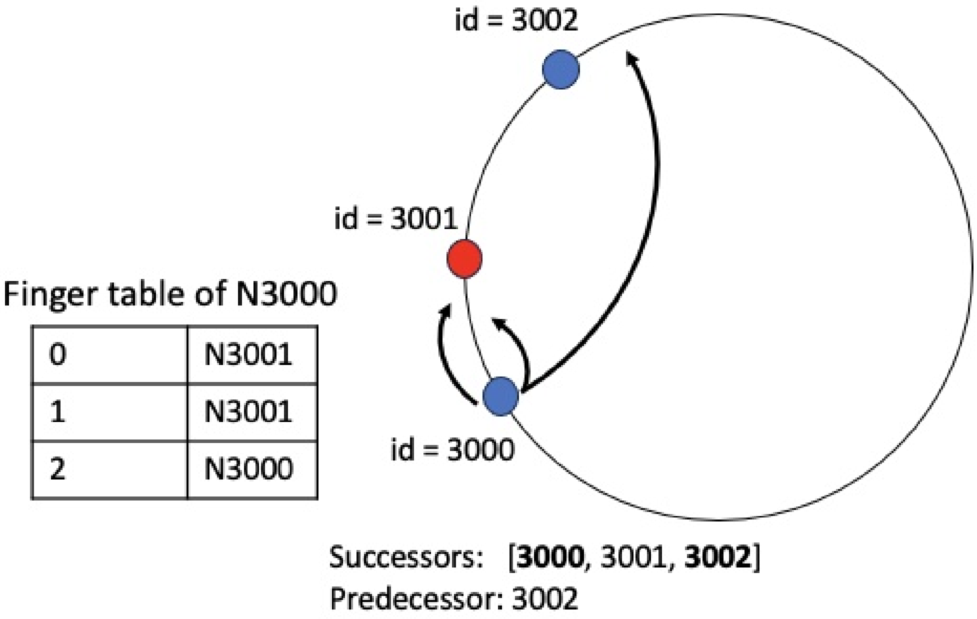
Finger table of N3000

0	N3001
1	N3001
2	N3000

Figure 3.1: Incorrect Chord

Notice how 3000's 0th and 1st finger points to 3001. Its 2nd finger skips node 3002 altogether and would hence point back to 3000. In the event of 3001 failing there would be no way of assigning a correct successor for 3000 as there is a lack of information on 3002's existence as the fingers have skipped 3002.

To counter this, we implement the Successor List as shown in Figure 3.2.



Finger table of N3000

0	N3001
1	N3001
2	N3000

Figure 3.2: Corrected version with Successor List

Again, in the event of 3001 failing the presence of 3000's Successor List provides information of 3002. Once 3001's death has been detected, 3000 can consult its Successor List to assign 3002 as its new Successor.

In our implementation, the maintenance of the Successor List takes place inside the stabilize function. To prevent race conditions, any operations on the Successor List happen only after all operations that update the Successor and Predecessor of the node are complete.

3.3.2 Query correctness

When a node queries for a DNS record of a specified host-name, it searches in this particular order:

1. LRU cache
2. Local storage
3. Chord Network
4. Hierarchical (Traditional) DNS

If the query is forwarded to the hierarchical DNS, the results are then added to the Chord network. The website string is hashed using the same hash algorithm for Node IDs in the Chord network. A correct successor for the hashed website is found, relying on the correctness of the underlying Node structure (as discussed in the previous section). Since the hash function is expected to be evenly distributed around the circle, the distributed load for storing DNS records in each node is amortized to be equal.

3.4 Scalability

The Chord protocol is scalable, and the cost of a lookup in Chord grows proportionally to the log of the number of nodes, making systems with a high number of nodes still feasible. The key to achieving such scalable lookup performance is the finger table, which was shown earlier in Figure 1.4

The schema of the finger table is such that, each node stores information on a small number of nodes rather than all nodes, reducing the storage requirements of each node. Furthermore, each node knows more about nodes that are closer to itself in the Chord ring. With these 2 properties, there is a high probability that the number of nodes contacted to find a successor of any arbitrary key k in a network with n nodes is $O(\log N)$, which is highly efficient, making the Chord protocol highly scalable in terms of key lookup.

3.5 Decentralization

Chord is a fully distributed protocol, where no node is more important than any other. Firstly, every node in the system contains approximately the same number of keys, through consistent hashing. For our key, which is the domain name of the website, we used SHA-256 as our hash algorithm. SHA-256 possesses several advantageous properties, making it well-suited for serving as our hash function. Most essentially, it displays an avalanche effect, where a slight change in the input, results in a significant change in output, which implies strong randomization [3] and allows the keys to be more evenly distributed among the nodes. This is especially relevant for domain names, as many domain names are very similar to each other. Additionally, each node can be queried and can be involved in the key lookup.

Chord is also resistant to attacks that centralised systems are vulnerable to, mainly denial of service(DOS) attacks. A DOS attack is a type of attack that aims to disable a server from servicing its clients, through attacks such as flooding the server with invalid data or sending an excessively high number of requests to slow it down [1].

The decentralized nature of the protocol means that there is no single point failure in our system, as other nodes can still handle the query and lookup even if 1 node were to fail or fall to such an attack. This improves the reliability of our system and makes it more fault-tolerant than other centralized protocols. Furthermore, consistent hashing also means that nodes can enter and leave with minimal disruption to the network. With high probability, when 1 node leaves the network consisting of k keys and n nodes, only $O\left(\frac{K}{N}\right)$ keys will be shifted. This is significantly more efficient than other centralized protocols, where a total of $O(K)$ keys might need to be shifted upon a centralized server failure. The different types of fault tolerance that our system is capable of tolerating will be elaborated on in the next section.

3.6 Fault Tolerance

The distributed system implementation handles multiple types of faults, all of which are listed below:

- Permanent Failures
- Planned/Voluntary Failures
- Intermittent Failures
- Some Byzantine Failures

3.6.1 Permanent Failures

Our implementation handles permanent failures through the replication of DNS records associated with a node in its next k successors, where k can be decided based on the intensity of fault tolerance required for the use.

Let us assume that p is the probability of a permanent node failure in our implementation, where $p \in [0, 1]$. If the replication factor of the DNS records is k including the original node itself, the probability of losing these records permanently would be p^k, reducing exponentially for a linear increase in k.

The DNS-Chord implementation periodically checks if the associated predecessor of a node is alive. In the case where a failure is detected, the replicated records are merged according to the new consistent hashing space on the chord ring, making the records still accessible in the case the node never reboots or has permanently failed.

3.6.2 Planned departures

Our system tolerance to handle permanent failures has inherently dealt with fault tolerance towards voluntary failures or planned departures in the distributed system. The mechanism to detect a failed node through periodic checks of the predecessor of a node allows us to detect a planned failure or voluntary departure.

We then adjust the distribution of records in the chord structure to reflect the distribution as stipulated by the consistent hashing mechanism.

3.6.3 Intermittent failures

There will be instances in a distributed system where a node might crash and instantly reboot due to some system faults or unrelated crashes. These crashes might be fatal when each node involved in the distributed system stores the metadata associated with the distributed system on volatile memory. The rebooting of these nodes will trigger a complete wipe-out of the metadata in the memory, treating the reboot as a new node joined into the distributed system.

Our implementation can handle these kinds of intermittent failures as well. We are able to persist the metadata on volatile memory through periodic writes to non-volatile memory in a node. At the time of reboot from a crash, the node will check for metadata written to non-volatile memory. If there is any existing metadata on the persistent storage, the node is able to restore its state to the instance before the crash.

If the time interval between the reboot and crash is short, this mechanism of dealing with this failure is efficient because it will not trigger a redistribution of the DNS records in the chord ring structure, saving network bandwidth between the nodes.

3.6.4 Byzantine Failures

Our implementation is also capable of handling certain byzantine fault scenarios. In the case of a byzantine failure, where one node refuses to respond to another node's DNS query, our replication factor of k makes sure that there are at least k copies of the records with $k - 1$ consecutive successors of the node with the primary copy.

In this case, we can retrieve the results despite the denial of service/malicious behavior from a byzantine node, by approaching any of its $k - 1$ successor node records.

Although the scenario discussed above does not fully cover all types of malicious faults, we propose an alternative implementation - one in which we trade off performance for increased fault tolerance.

First, instead of querying the node that is stipulated to contain the record and retrieving the list of IPs, we may introduce a proxy step, where the requester retrieves the successor list of the node that must be storing the query. The requester may then proceed to query all three nodes simultaneously to retrieve at most k responses, where k is the replication factor. With k replicas, the system can have a maximum of $\frac{k-1}{3}$ faults. Which means in order to receive a super majority of responses, it needs to receive at least

$$(2 * \frac{2k+1}{3}) + 1$$

matching responses. Therefore, to settle on a non-malicious response, the node must receive at least $\frac{2k+1}{3}$ matching responses. by ensuring that the querying node receives at least $\frac{2k+1}{3}$ matching responses containing digests of website IPs, we can overcome byzantine failures.

3.7 Limitations and Future Work

3.7.1 More Byzantine Fault Handling

Our current design can handle only a specific case of byzantine fault, where we define a malicious node as one that gives us a response that is different from the true response. As mentioned in previous sections, accounting for all kinds of byzantine faults may also affect the performance of our system. Thus, we hope to find a balance between the ability to tolerate different kinds of faults without affecting the performance of the system.

3.7.2 Improved Reaction Time

Currently, it takes a while for our system to detect node failures (approximately 10 seconds), especially when the number of nodes is high. In the future, we hope to improve the system's reaction time to node failures, even as the number of nodes increases, to further enhance the robustness and scalability of the system.

3.7.3 User Interface

At the moment, all the inputs are via the terminal and our outputs are also text-based. In the future, we hope to design and implement a user-friendly graphical interface that allows users to perform DNS queries on our system. A visual representation of our system, such as the ring structure, may also be displayed to the user, and interactions with the system(e.g. node departures, chord network connections, etc.) can be performed directly through the graphical interface.

3.7.4 Variable IP address handling

The current implementation in the repository is intended for testing with local usage within subnets, and as such hashes the IP addresses of nodes to create node IDs. This allows us to spin up multiple nodes within a system using different port numbers as the differentiator when it comes to hashing. The storage also gets differentiated when multiple nodes are spun up within a system, which is feasible for testing purposes. However, a more correct implementation would make use of MAC addresses, as they remain static, regardless of what IP gets allocated to the machine by its nearest DHCP server.

Chapter 4

Experiments/Evaluation/Tests

In the evaluation of the performance of our distributed system, we made use of a data-set obtained from **kaggle**. The data-set needed some level of preparation to be passed into our chord implementation. It consisted of unnecessary columns, which were removed in the final version. For this section, let us call the set of queries used for evaluation as q, where q_i denotes the i^{th} query.

Performance can be measured in mainly two ways:

- **Message complexity:** Can be measured in terms of hop count.

$$hops(i) \coloneqq number\ of\ hops\ for\ each\ query\ q_i$$

- **Time complexity:** The amount of time taken to execute a queries.

$$time(n) \coloneqq time\ taken\ to\ execute\ n\ queries$$

4.1 Message complexity - Hop Count Analysis

We devised a comprehensive strategy to evaluate the hop count against legacy DNS. We varied both, the number of nodes in the network, as well as the number of queries being passed. Testing with 5, 7 and 10 Nodes we obtained the average hop count for a varying number of queries performed. The number of hops represent the number of nodes the query passed through before reaching the owner of the queried DNS record. Finally, we also measured hop counts taken for DNS retrievals on legacy DNS structure for 1000 queries.

We can formalize the average number of hops in a z node system in a chord network as $hops_z$:

$$hops_z \coloneqq \frac{1}{10}\sum_{k=1}^{10}\frac{1}{100k}\sum_{i=1}^{100k} hops(i)$$

For our evaluation, we found $hops_5$, $hops_7$ and $hops_{10}$, which has been plotted in Figure 4.1.

Our results showed that our system significantly outperforms legacy DNS in terms of average hop count. This can be mainly attributed to two reasons:

1. The chord network proposed in this implementation is formed within a subnet, instead of the global scale. This means that the total number of nodes in the chord network is significantly less than the total number of routers that a regular DNS query might have to pass through.
2. The maintenance of the finger table along with the consistent hashing method makes it easier for nodes to reduce their search space and offload queries in case the location of the query falls outside the range of nodes covered in the finger table. Moreover, the caching mechanism, along with the lightweight memory storage guarantees that the speed of execution within the subnet is extremely fast at the local level.

Furthermore, the results also showcase the scalability of our system, as the average hop count only increases slightly even when the number of nodes doubles.

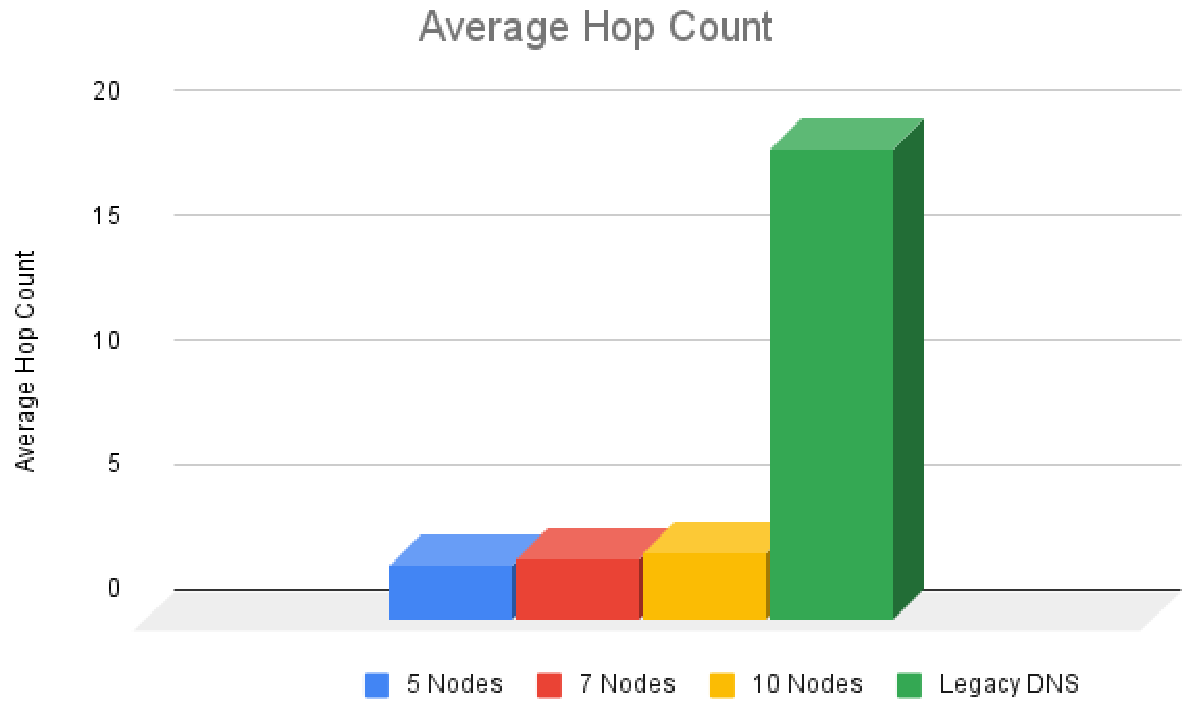


Figure 4.1: Average Hop Count

4.2 Time complexity analysis - Time taken for DNS query

Our second evaluation criteria involved measuring the time taken for the z nodes in the system to successfully retrieve n DNS queries, where we vary both z as well as n. For our analysis, we varied z in steps 5, 7, and 10, and varied n in steps 100, 200, 300… 1000. The results obtained for 5 nodes for n queries were averaged to obtain one entry. Likewise, a similar approach was undertaken for $z = 7$ and $z = 10$.

This can be formalized as follows.

$$time_z \coloneqq \{ \; time(100k) \mid k \in [1, 10] \; \}$$

We plotted sets $time_5$, $time_7$ and $time_{10}$.

For the legacy DNS system, a similar approach was taken, by varying the number of queries only. The number of nodes is inconsequential in traditional DNS, as we are only concerned with the overall performance of the DNS system.

The results obtained have been summarised in Figure 4.2. In conclusion, our system outperforms legacy DNS, due to the storage and caching mechanism in our system, leading to efficient lookup times. However, with 10 nodes in our system, the total query time increases slightly, showing the potential limitations of such a system to be deployed in a wide-scale environment, which is why we believe the implementation of a system is more beneficial within subnets, where the number of nodes is smaller. Global implementation of the chord system proposed here would prove to be infeasible, as the number of processors to pass through before finally getting to the authoritative processor would far exceed that of traditional DNS systems.

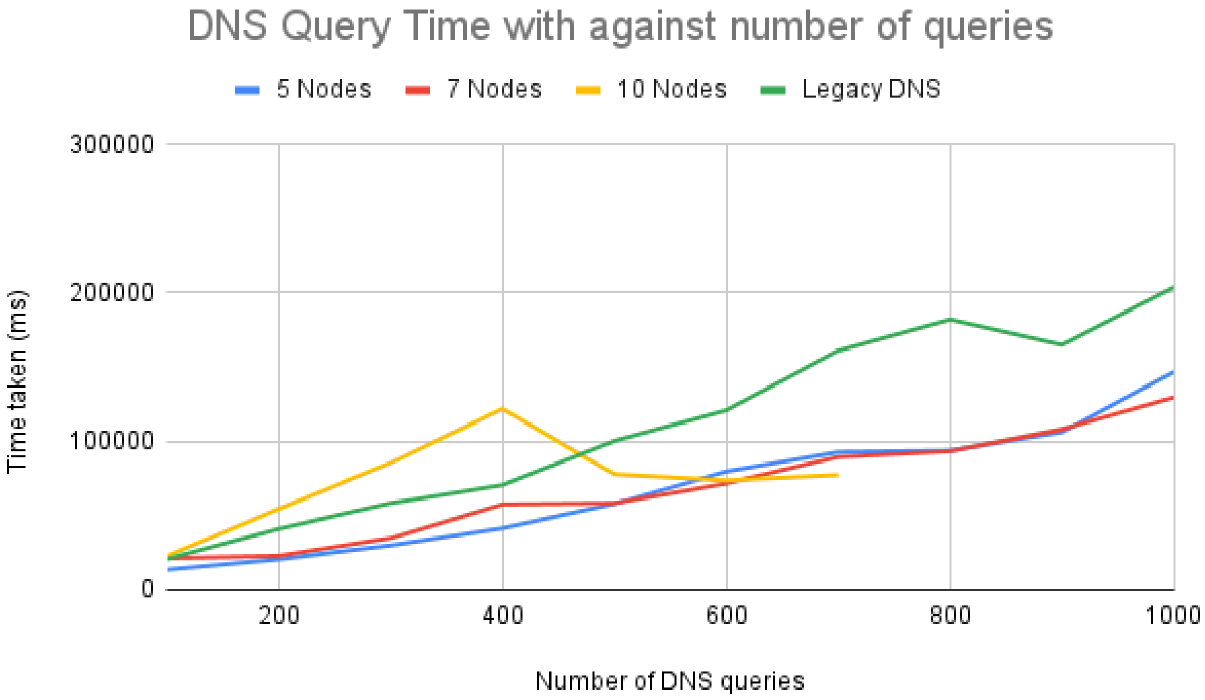


Figure 4.2: Graph showing the total time taken as the number of queries and nodes increase

Chapter 5

Conclusion

In summary, we have successfully designed and implemented DNS built on top of the Chord Protocol, and showed that there is an improvement in performance in terms of message complexity, as well as average time taken to retrieve n queries. Our system has been designed and tested to be correct and scalable, with a certain degree of fault tolerance. In the future, we hope to increase the fault tolerance and scalability of our design. Lastly, we would like to thank Professor Sudipta for his unwavering support and guidance to the team throughout this project.

Bibliography

[1] G. Carl, G. Kesidis, R. R. Brooks, and Suresh Rai. Denial-of-service attack-detection techniques. *IEEE Internet Computing*, 10(1):82–89, Jan 2006.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb 2003.

[3] D. Upadhyay, N. Gaikwad, M. Zaman, and S. Sampalli. Investigating the avalanche effect of various cryptographically secure hash functions and hash-based applications. *IEEE Access*, 10:112472–112486, 2022.

[4] O. van der Toorn, M. Müller, S. Dickinson, C. Hesselman, A. Sperotto, and R. van Rijswijk-Deij. Addressing the challenges of modern dns: a comprehensive tutorial. *Computer Science Review*, 45:100469, Aug 2022.

[5] Pamela Zave. How to make chord correct. 2015.