50.041 Distributed Systems and Computing

Final Report

Chord-Powered TinyURL

Group 3

Abram Tan Jia Han 1005057

Yong Zheng Yew 1005155

Lim Rui-Chong Anthony 1005264

Charlotte Ng Si Min 1005320

Ankita Sushil Parashar 1005478

Table of Contents

1. Introduction and Problem Formulation **3**
2. Overview of the System **4**
3. Design **7**
3.1 Features 7
3.2 Correctness/consistency 7
3.2.1 Definition of correctness 7
3.2.2 Correctness of fundamental Chord functionality 8
3.2.2.1 Storage Use Case 8
3.2.2.2 Retrieval Use Case 12
3.3 Fault tolerance 14
3.3.1 Permanent Faults 15
3.3.2 Intermittent Faults 16
3.3.3 Voluntary Leaving 16
3.4 Scalability 17
3.4.1 Finger Tables 17
3.4.2 Cache 18
3.5 Limitations 19
3.5.1 Cache Eviction Policy 19
3.5.2 Global Cache Implementation 19
3.5.3 Data Persistence 19
4. Experiments/Evaluation/Tests **20**
4.1 Experiment 1 20
4.2 Experiment 2 21
5. Concluding Remarks **23**
6. Future Work **24**
7. Appendix **25**

1. Introduction and Problem Formulation

TinyURL is a URL shortening service, taking long, unwieldy URLs and turning them into shorter, more aesthetically pleasing URLs. These shortened URLs are used in place of longer URLs, and they serve to redirect users to the original URL when accessed.

With the scale of the internet growing rapidly, and the increasing popularity of TinyURL among businesses and users worldwide, it is important to ensure that the service to store the mappings between the long and short URLs has to be correct, scalable and reliable. As the TinyURL service expands, it becomes highly impractical to store the mapping records on a centralized database server. This centralized architecture would have significant limitations in its performance, scalability and reliability.

In the light of distributed systems, the Chord Protocol offers solutions to these challenges, ensuring correctness, scalability and reliability. Chord is a Scalable Lookup Protocol that uses distributed hash tables. It uses a distributed hash function, distributing the key-value pairs, where the key is the ShortURLs and value is the LongURLs evenly across many nodes.

In this project, we aim to develop a scalable and fault-tolerant URL shortening service modelled after TinyURL, with the Chord Protocol as an underlying implementation.

To access the Github repository, refer here.

2. Overview of the System

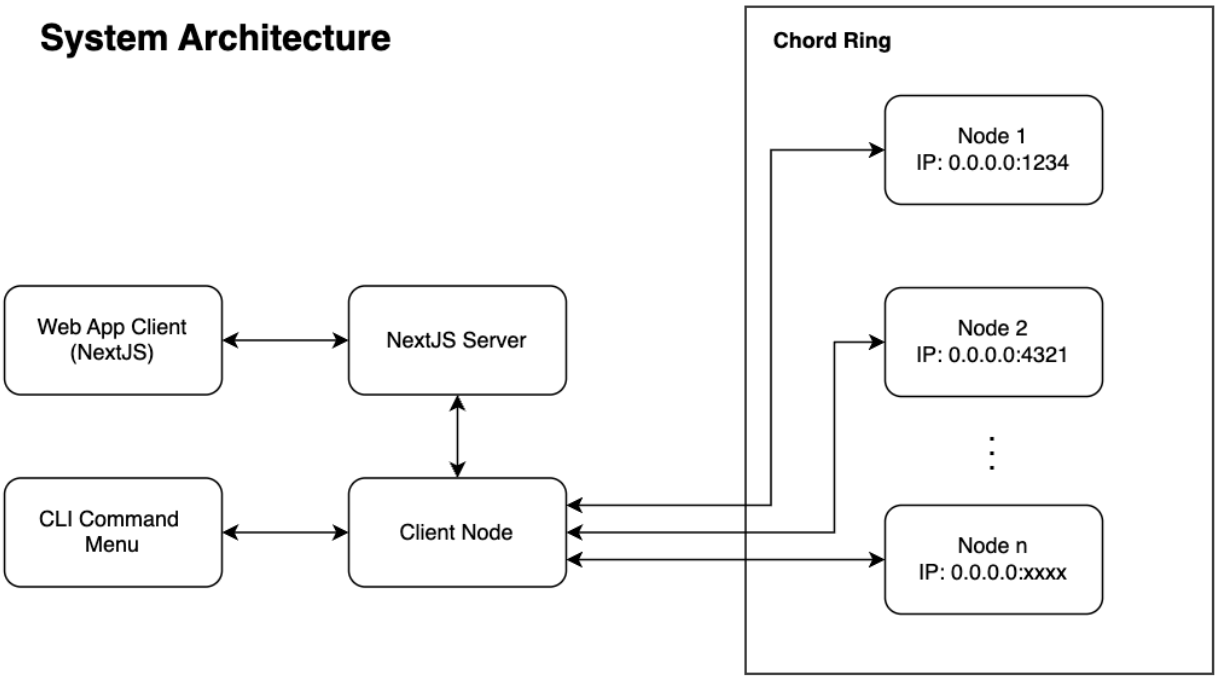

Figure 1: Overall system architecture of the Chord URL shortening system

The Chord URL shortening service comprises of 4 key parts, a web app for user interactions, a command-line interface (CLI) for interacting with and monitoring the distributed backend, a client node that serves as a single entrypoint for clients to the distributed backend, and finally the core of the project, the Chord ring nodes. These systems were architected to interact with each other with the overarching goal of good user experience, even when massively scaled and put through the paces of real world faults.

At the core of the Chord URL shortening service are 2 key features, the ability for users to store LongURLs, and by supplying a ShortURL, to be able to retrieve the original LongURL.

For the STORE functionality, the user sends a LongURL to the system, which is hashed to become the ShortURL. Consistent hashing is then used to determine which node is responsible for the ShortURL-LongURL key-value pair. The key value pair is then stored into that respective node. In a real-world scenario, the user gets feedback on what his ShortURL is. To the user, this is a mapping between the LongURL and the ShortURL, which he can then use in the future. This store feature can be accessed via 2 ways, the CLI command menu, or via a HTTP POST request to /store which is handled by the web app.

Store URL Flow

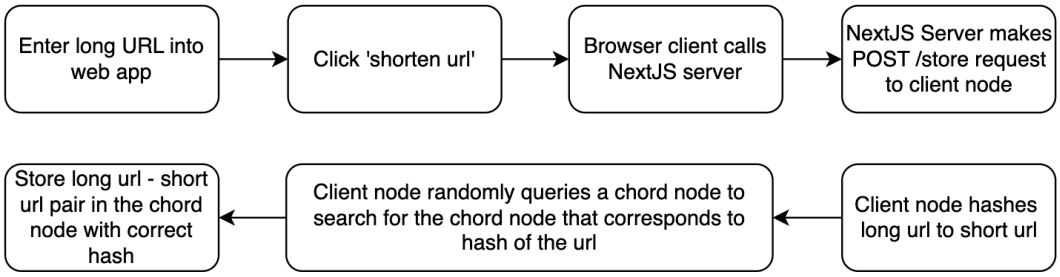

Figure 2: Process flow of storing a new LongURL

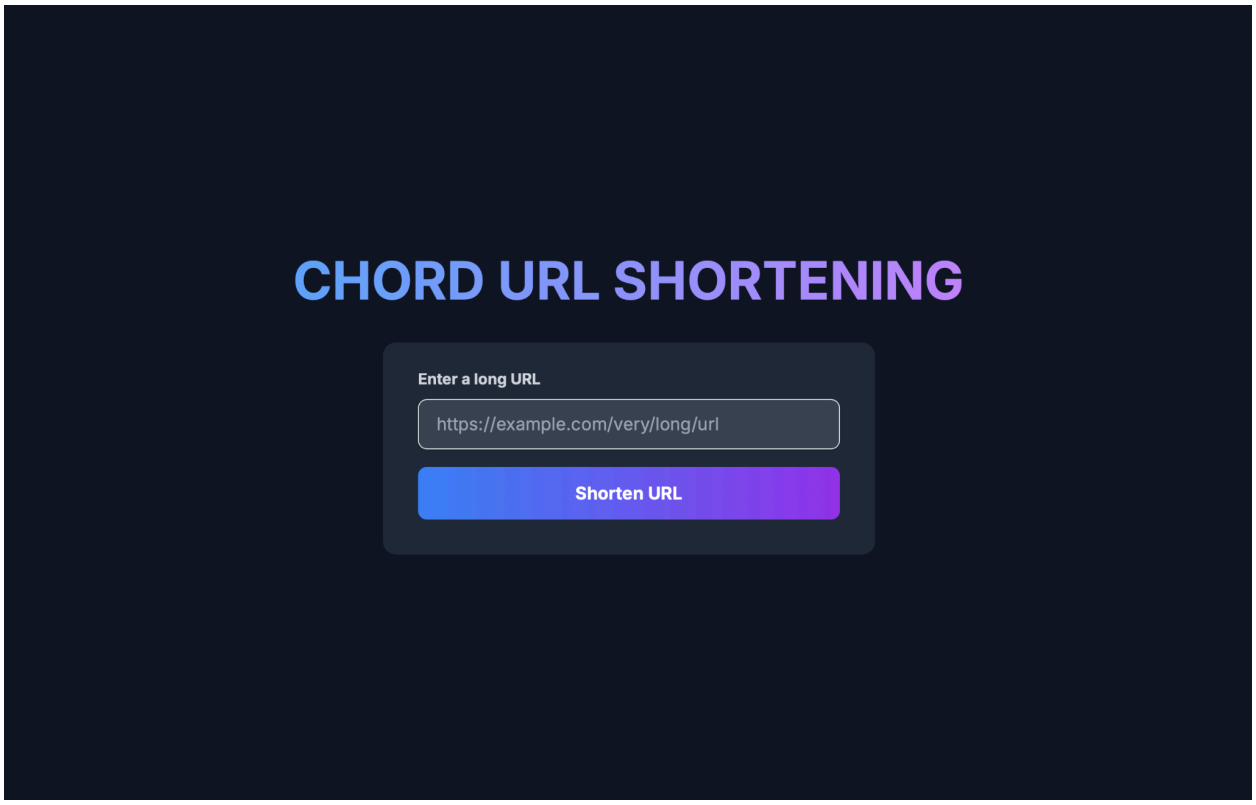

Figure 3: Screenshot of the web UI that a user would see and interact with

For the RETRIEVE functionality, the user knows his ShortURL. The user sends his known ShortURL to the system, which is then used to query the Chord Ring to find the node responsible for the mapping. The LongURL is retrieved and returned to the user. However, once a LongURL is retrieved, it is not actually directly sent back to the user. All the user needs to do is to enter the ShortURL in the browser. What happens behind the scenes is that upon receiving the LongURL, the web app automatically redirects the user to the original LongURL. The RETRIEVE function can be accessed via the CLI command menu or a HTTP GET /retrieve request which is abstracted away by the web app.

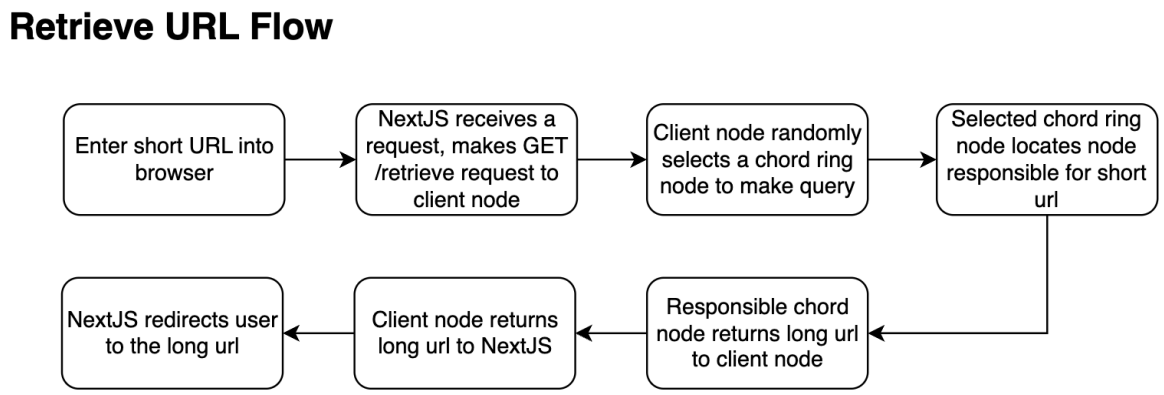

Figure 4: Process flow of retrieving a LongURL given a ShortURL

These are the 2 main functionalities implemented, which utilizes the underlying Chord Protocol as a distributed hash table service. In our simulation, nodes are represented as goroutines, and communication between nodes is done through RPC (the global array NodeAr is for debugging and interactivity purposes). In the next section, the details of the design and how the Chord URL shortening system works is discussed.

3. Design

3.1 Features

The Chord protocol is a structured peer-to-peer distributed hash table (DHT) designed to provide scalable and efficient lookups in a decentralized network. It organizes nodes in a logical ring structure and uses consistent hashing to assign keys and data to specific nodes, ensuring load balancing and minimal redistribution of keys when nodes join or leave.

Our implementation of the TinyURL running on Chord has the following key features:

Feature	Description
Consistent hashing	Ensures even distribution and minimizes redistribution of data, even when nodes join or leave unpredictably. Increases scalability.
Finger tables	Nodes at key intervals in the ring are memoized, to make searching the ring much more efficient with large numbers of nodes. Increases scalability.
Successor lists	Each Chord node has duplicates of its data in some of its successors to provide redundancy for fault tolerance. Set by the constant 'REPLICAS'.
Caching	Temporarily stores frequently accessed URLs to reduce lookup latency and improve system performance.
Involuntary node leaving	When a node leaves, the surrounding nodes eventually repair the ring and take over the data that the departed node had.
Voluntary node leaving	If a node knows it wants to leave, we provide a way for it to more efficiently initiate data transfer than in the involuntary node failure case.

Table 1: List of the Chord URL shortening service's key features

3.2 Correctness/consistency

3.2.1 Definition of correctness

Correctness can be understood as the ability to achieve our core functionality as defined. The core functionality of Chord in our application stems from our use cases. Namely, there are two:

- Storage use case:
    - Receiving a LongURL from a client
    - Generating a ShortURL
    - Storing the ShortURL-LongURL pair

- Retrieval use case:
    - Receiving a ShortURL from a client
    - Retrieving the LongURL

3.2.2 Correctness of fundamental Chord functionality

3.2.2.1 Storage Use Case

Receiving a LongURL

To be correct, our implementation needs to accept a LongURL as input. This is achieved in our implementation since the store process starts when a random node receives a CLIENT_STORE_URL message through ClientSendStoreURL().

Generating a ShortURL

From the LongURL, the service hashes the LongURL, and the hash to be the ShortURL.

Storing the ShortURL-LongURL pair

To be correct, our implementation needs to store a (ShortURL, LongURL) pair. On top of that, since we are using the Chord protocol, this pair must be stored by the correct node.

Design of the URLMap

Each node stores key-value pairs in its UrlMap, which in Go types, is a nested map:

- Key: IP address of node who owns data
- Value: Map of
    - Key: Short URL
    - Value: Long URL

The reason for the first layer of the map is because:

- In our implementation which includes successor lists, every node stores some data that belongs to other nodes.
- When faults occur, each node must remember which sections of data belong to itself, and which belong to previous nodes. This is so that when rebuilding successor lists and propagating data for redundancy purposes, nodes will avoid an infinite chain where data is passed around endlessly to everyone. Instead, data from a failed node will only be passed to the N nodes in its own successor lists.

In Chord, the correct node to store a key-value pair is determined by the ID assigned to both the pair and the node by a consistent hashing algorithm. Chord uses a variant of consistent hashing that makes use of the assumption that the hash function used, though deterministic, is intractable to reverse (this property is also called the “standard hardness assumption”). Chord’s

consistent hashing should generate a ring-shaped ID space (an *identifier circle*), onto which both key-value pairs as well as nodes are mapped. Each pair should be stored by the first node which follows it in the identifier circle.

Thus, our requirement for correctness is:

- Correctness of identifier circle:
    - **CR1:** Our hashing function must take as input either a key-value pair or a node, and output an id
    - **CR2:** The domain of the output should be finitely bounded
    - **CR3:** The domain of the output should have a cyclic ordering.
- Correctness of mapping from input onto identifier circle:
    - **CR4:** The mapping should be deterministic, in order to be able to reliably retrieve a stored key-value pair.
    - **CR5:** The mapping should produce IDs in a reasonably uniform distribution for random input.
    - **CR6:** The chance of two mappings clashing (i.e. hash collision) should be negligible.
- Correctness of assigning key-value pairs to nodes
    - **CR7:** Each key-value pair should be stored by the first node which follows it in the identifier circle.

Our implementation closely follows that given in the Chord paper. Namely, in our GenerateHash() function, we:

- Take as input either a ShortURL, or a nodeID
- Use the SHA 256 hash function to hash the input
- For performance, truncate the hash
- Convert the hash to an integer
- Modulo the integer by the total size of the ID space. This is our output.
    - Total size of the ID space is expressed as 2^m, where m (which is a global constant) is the number of bits needed to express an id.

These output IDs are then compared using the InBetween() function. The goal of this function is to take two hashes, denoting the start and end points clockwise (i.e. increasing integer values) on the identifier circle, and then check if a third hash, the target, lies in between them clockwise. The algorithm is as such:

- We take as input 3 hashes: One for the beginning, one as the target, and one for the end.

    - Recall that hashes in our implementation are really just integers. When we say one hash is larger than the other, or one hash > another hash, we mean this in the normal integer sense.
- If the start hash < the end hash:
    - Recall that by definition we intend to check the part of the circle clockwise from start to end, so in this case we know that we are considering a segment of the circle that does not pass through the zero/max point.
    - Thus, return true if start < target AND target < end, and false otherwise.
- If the start hash > the end hash:
    - We know that we are considering a segment that passes through the zero/max point.
    - Thus, the target can either be in:
        - The part of the circle in between the start and the zero/max point, or
        - The part in between the zero/max point and the end.
    - Thus, return true if either start < target OR target < end, and false otherwise
- We have a few, less important edge cases that we will not go into detail about:
    - If start == end, this is for the case where there is only one node in the ring; in which case any added id will be in its sector of responsibility. Thus, always return true.
    - In our implementation, we have an extra parameter includeUntil, which makes inequality checking inclusive for the “end” portion.

The general flow of storing a key-value pair in our implementation of the Chord network goes as such:

1. A random starting node receives the request from a client to store the key-value pair
2. The starting node finds the node which is the immediate successor of the pair.
3. The starting node then directly sends the pair to the immediate successor node to store
    - Edge case: If the immediate successor is itself, then perform the storing.

Steps 2 and 3 are contained within our function StoreURL().

In order to show satisfaction of **CR7**, we must further examine how step 2 above works. The function called by the starting node in order to find the immediate successor is findSuccessor().

Inputs:

- startingNode: the node that is searching for the immediate successor
- targetID: the hash of the key-value pair that we wish to store

Output: the ip address of the first node that follows (clockwise) the targetID

How it works:

- If targetID is in between startingNode and startingNode's immediate successor node:
    - We know that there is no other node between startingNode and targetID
    - We want the first node that succeeds targetID.
    - Thus, return the ip address of startingNode's immediate successor. This is the base case.
- Else:
    - Use the finger table of startingNode to find the closest memoized node to the target ip address. This is done by calling closestPrecedingNode().
    - Recursively call findSuccessor() on that node through RPC, and return the reply as output.
    - Some error handling: If we find out that that closest memoized node doesn't exist, then delete that node from the finger table and just retry findSuccessor().

Note that the above function is dependent on closestPrecedingNode(). Let us examine how it works.

Inputs:

- startingNode: the node calling this function
- targetID: the hash of a key-value pair

Output: the ip address of the furthest away (clockwise from startingNode) memoized node that still precedes the targetID

How it works:

- Recall that the finger table of startingNode represents an array of memoized nodes and their hashes (in our implementation, hashes are not explicitly stored, but are calculated on demand).
- We know that each node in the finger table is either preceding or succeeding the targetID.
- We know that nodes in the finger table are sorted in such a way that as we go down the list, each node is further and further away (clockwise) from the startingNode.
    - Strictly speaking, we can have duplicates in the finger table, so it is more accurate to say that each node is at least as far away from the startingNode as the previous node in the finger table.

- Thus, we iterate through the finger table in **reversed** order. We return the first node we come across that precedes the targetID. This is the farthest node from the startingNode that still precedes the targetID.

Achieving CR1: The input of GenerateHash() is either a ShortURL, or a nodeID, which correspond to our key-value pairs and our nodes respectively:

- Our key-value pairs are (ShortURL, LongURL) pairs. For convenience, we choose only the ShortURL portion to generate hashes.
- Each node is initialized with an arbitrary but unique nodeID.

Achieving CR2: Our output is int64. The modulo by 2^m puts upper and lower bounds on the output domain:

- An upper bound of 2^m-1 due to the choice of 2^m
- A lower bound of 0 since modulo output is non-negative by definition.

Achieving CR3: IDs are only compared using the InBetween() function, which as described, effectively implements a cyclic ordering.

Achieving CR4: SHA-256 is used in GenerateHash(), which is deterministic.

Achieving CR5: SHA-256 is generally taken to be reasonably uniform in output distribution.

Achieving CR6: Our parameter of m can be arbitrarily set to a large enough number such that, with **CR5**, the chances of hash collisions are small since the total output domain size is large.

Achieving CR7: We have already seen the process that stores a key-value pair, and explained how and why it works. No matter which node in the network receives the key-value pair, it then calls StoreURL() to find the first node which follows the key-value pair in the identifier circle, and stores it there.

3.2.2.2 Retrieval Use Case

Receiving a ShortURL

The retrieval process in our implementation starts when a node receives a CLIENT_RETRIEVE_URL message, which contains the ShortURL corresponding to the (ShortURL, LongURL) pair we wish to retrieve.

Retrieving the LongURL

The process for retrieving the LongURL is as such:

1. A random starting node receives the CLIENT_RETRIEVE_URL message containing the ShortURL.
    a. Recall that the hash for each (ShortURL, LongURL) pair is solely generated from the ShortURL field.
    b. Thus, just from the ShortURL, the starting node calculates the hash for the entire pair.
2. We already know due to **CR7** that the (ShortURL, LongURL) pair must have been stored by the first node which succeeds it in the identifier circle.
3. The starting node calls retrieveURL(), which works very similarly to storeURL().
    a. retrieveURL() first calls findSuccessor() to find the first node which succeeds the (ShortURL, LongURL) pair.
    b. It then sends that node a RETRIEVE_URL message. That node should reply with a message containing the (ShortURL, LongURL) pair.
        i. Edge case: if the node returned by findSuccessor() happens to be the starting node, then just return the pair directly.

As mentioned earlier, all ShortURLs are unique due to use case assumptions, so assuming no hash collisions, the output ids will be unique as well.

Although our key-value pair is a (ShortURL, LongURL) pair, for convenience, we use just the ShortURL portion to generate the id for a pair.

Initializing the Ring

In our simulation, if a node is initialized when no other node exists, it will call CreateNetwork(), which just sets its predecessor to nil and its successor to itself.

Otherwise, when a node is initialized, it is given the IP address of a random existing node in the ring. It will then call JoinNetwork(), which finds the correct place for the new node within the network. Since nodes and key-value pairs share the same ID space, It does this by using FIND_SUCCESSOR and setting the new node's successor to the resulting node.

In order to fully integrate the new node into the ring, we need:

- The correct successor of the new node (let us call this existingSucc) recognizes the new node, sets its own predecessor to the new node, and transfers keys to the new node which should belong to it.
- The correct predecessor of the new node (let us call this existingPred) recognizes the new node and sets its own successor to it.

For both of these, we rely on the Maintain() function, which is periodically called by every node. Maintain() calls fixFingers(), stabilise(), checkPredecessor() and MaintainSuccList(), which are important to fault tolerance, which overlaps with node joining. Here, we focus mainly on stabilise().

The joining node eventually calls Maintain(), which calls stabilise(), which in turn sends a message to the existingSucc to call Notify(). Notify() causes existingSucc to realize the joining node should be its new predecessor, and set its own predecessor correctly as well as transfer some data to the new node through the Keys field in a NOTIFY_ACK message.

The existingPred also eventually calls Maintain() which calls stabilise(). This causes it to realize that existingSucc's predecessor is no longer itself, but the new node. existingPred will check that the new node is in fact in its correct ID place, and will set it as its own successor.

In this way, the new node is fully joined into the ring.

3.3 Fault tolerance

In order to support our ability to achieve the core use cases, we have certain requirements on the integrity of the system:

- **FT1:** If a node departs, that data must eventually be taken over by another node
    - "Taken over" means that that node will serve requests for that data
- **FT2:** If a node departs, the ring structure eventually will self-repair.

In order to fulfill these requirements, every node maintains a list of successors to which its data will be duplicated. In the case that a node faults, these successors will recover the data in a procedure described later.

This list is constructed using MaintainSuccList(), which is under Maintain() and will thus be called periodically. When a node calls MaintainSuccList(), it sends GET_SUCCESSOR_LIST to its successor. If the successor responds with a list with elements 1 to n, the node constructs its own successor list by prepending its immediate successor to elements 1 to n-1 of the received list.

If the successor does not respond, the node will select the next node in its own existing successor list and set that as its successor. We then rely on the next time the node calls MaintainSuccList() to construct the successor list (or defer to the next MaintainSuccList() call and so on).

Once the list is constructed, to each node in that list, the node sends SEND_REPLICA_DATA with a copy of its data.

How We Simulate Faults

We simulate faults in a node by enabling a node's FaultFlag. This causes the node to not respond to all RPC calls with other nodes and stop running Maintain(). For all intents and purposes, the node is dead. The node will still show up in the SHOW command in the interactive menu, but it is essentially invisible to the other nodes, who will repair the ring and take over its data as needed. We simulate the recovery of such a node by disabling its FaultFlag, which returns it to normal and initiates the normal ring repair procedure among the nodes.

3.3.1 Permanent Faults

In this scenario, a node faults and never responds again or sends messages to any other node.

Achieving FT1: Every node periodically calls Maintain(), which calls checkPredecessor() to ping its predecessor. If its predecessor has faulted, an ACK is not received; the calling node sets its predecessor value to NIL and absorbs the keys that were stored under its predecessor's IP (since it would be part of its predecessor's successor list and therefore received replica data for precisely this purpose).

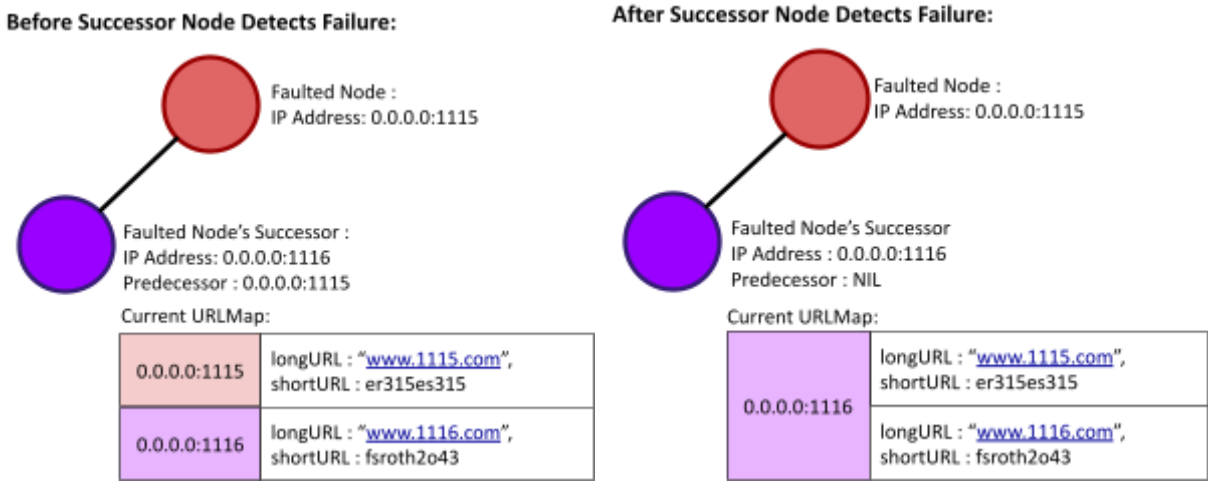

Figure 5: Update of keys upon failure detection

Achieving FT2: On the other hand, when the faulted node's predecessor runs stabilise() and does not receive an ACK from the faulted node, it then sends a getPredecessorMsg to the next node in its successor list. Here, the successor list helps to maintain the ring as it provides the faulted node's predecessor with the faulted node's successor. It sends down a

getPredecessorMsg and sets its new successor. The faulted node's successor also updates its new predecessor.

In this way, the successor list achieves the two requirements for fault tolerance stated earlier. By doing so, the protocol increases robustness and ensures that lookups remain correct even in the event of multiple simultaneous failures, significantly reducing the likelihood of incorrect lookups and strengthening the fault tolerance of the Chord ring.

In summary, the successor list is a critical feature of the Chord protocol, enhancing its ability to handle both permanent and intermittent faults. It ensures that the system remains robust, maintaining its correctness guarantees and enabling reliable operation even in challenging scenarios involving simultaneous node failures.

In the menu, this situation is simulated using the FAULT command.

3.3.2 Intermittent Faults

An intermittent fault is when a node faults, and then recovers. It is hard to distinguish from a permanent fault since when a node fails, we do not know if it will rejoin, so it is treated exactly the same when its fault is detected by its surrounding nodes, including data recovery by its successor.

Once operational again, the node is treated as though it is rejoining the ring, ensuring it adheres to the same correctness guarantees as any newly joining node. During this process, the node performs necessary stabilization steps, such as updating its predecessor and successor pointers and ensuring that it has accurate routing and key information. These intermittent faults are managed at a high level using mutex locks and unlocks within the FAULT and FIX methods, ensuring that the ring's stability and fault tolerance capabilities are maintained during such interruptions. This approach ensures minimal disruption to the ring's structure while guaranteeing correctness upon recovery.

In the interactive menu, faulting is simulated using FAULT, and recovery by FIX.

3.3.3 Voluntary Leaving

Technically, since the procedure for intermittent faults achieves our requirements for fault tolerance, we could have made it such that a voluntary leaving node will simply stop responding or contacting nodes in the ring. However, this relies on the next periodic Maintain() calls for eventual recovery.

To increase reliability, when a node **voluntarily** leaves the Chord ring, it follows a structured process to ensure the network's integrity. The process begins with key transfer, where the leaving node uses the Node.Leave() method to send its keys to its successor via an RPC call, ensuring data continuity. Next, the leaving node updates its successor using the voluntaryLeavingSuccessor(keys, newPredecessor) method, providing the successor with the transferred keys and instructing it to update its predecessor pointer to the leaving node's predecessor. Simultaneously, the leaving node notifies its predecessor to update its successor list. This is achieved through the Node.Leave() method, which sends an RPC call containing the last node in the leaving node's successor list. The predecessor processes this update using the voluntaryLeavingPredecessor(sender, lastNode) method, adjusting its successor list to reflect the departure accurately. Finally, the node sets its FailFlag to true within the Node.Leave() method, effectively "freezing" itself and exiting the ring. This coordinated use of methods ensures a seamless departure while preserving the Chord ring's structure and data continuity.

In the menu, this situation is simulated using the DEL command. This also triggers the FaultFlag, but unlike FAULT, it initiates the voluntary leaving procedure and is not intended to be recoverable using FIX.

3.4 Scalability

3.4.1 Finger Tables

To accelerate lookups of key-value pairs, Chord maintains additional routing information in the form of a routing table called the finger table. The ith entry in the table at node n contains the identity of the first node s that succeeds n by at least 2^{i-1} on the identifier circle. A finger table entry includes both the Chord identifier and the IP address of the relevant node.

Each node has finger entries at power of two intervals around the identifier circle. As such, each node can forward a query at least halfway along the remaining distance between the node and the target identifier. Therefore, the number of nodes that must be contacted to find a successor in a N-node network scales logarithmically, by O(logN). This presents much improvement over an iterative chain of RPC calls which visits every node on the circle, which has a complexity of O(N).

Finger tables are constructed and maintained using fixFingerTables(), which uses FindSuccessor() to periodically update each entry of the finger table. In this way, we tolerate node joins and leaves, including faults.

3.4.2 Cache

In addition to the Finger tables, our TinyURL application incorporates a caching layer that makes the protocol further scalable. The cache is implemented on the storage nodes where frequently accessed LongURLs are stored temporarily. When a query is made using a ShortURL, the protocol will first check the cache for the respective LongURL before starting an RPC call to another node in the protocol.

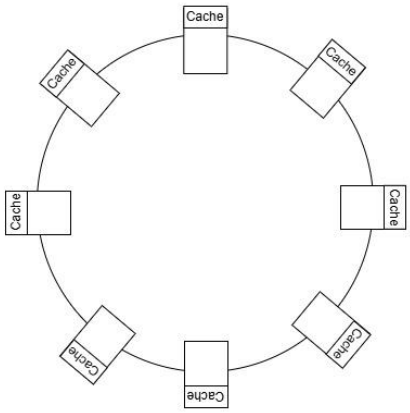

Figure 6: Cache in every storage node

Our implemented protocol goes through a series of steps as follows: Firstly, when a query is made with a ShortURL, the RetrieveURL function identifies the primary node responsible for the key using the FindSuccessor function. If the current node and primary node are the same, it would directly check its UrlMap for the ShortURL and return the corresponding LongURL if available. If not found or if the primary node and current node are not the same, the querying node will check its local cache which is stored within its UrlMap under a CACHE hash key. If both these ways fail to recover the LongURL, then the protocol sends an RPC RETRIEVE_URL message to the primary node identified previously which would return the LongURL and timestamp. A timestamp is also associated with the ShortURL in the case of conflict resolution. Upon receiving the LongURL and timestamp from the primary node, the timestamp is compared to the timestamp recorded in the cache. If the timestamp in the cache is outdated comparatively, then the cache is updated with the newer data to ensure consistency.

By implementing this multi-layered lookup mechanism, the protocol ensures a balance between reducing latency and maintaining consistency. The cache reduces the number of inter-node RPC calls for frequently accessed ShortURLs, lightening the load on the Chord network and improving query response times, especially under high traffic or with a growing number of nodes.

Technically, the redundant data contained by successor list nodes can also be easily used as a cache with just a few lines of code. A draft version of this was implemented, but due to time constraints, was not able to be thoroughly tested before the experiments were run. Thus, it is not in the final submission.

3.5 Limitations

3.5.1 Cache Eviction Policy

Currently, the cache implementation uses a very simple LRU policy based on timestamps and cache size. While it works, it lacks the sophistication and proper handling of complex caching needs. Moreover, the caching is stored in memory, limiting the scalability when dealing with larger amounts of data and could affect the cache hit ratio.

3.5.2 Global Cache Implementation

The cache entries are local to every node without any coordination between nodes, leading to redundant lookups and cache usage. Designing a distributed cache system was too complex given the focus of the project was the Chord protocol.

3.5.3 Data Persistence

The URLs and data is stored in memory. In the case of node failures, the system might be vulnerable to data loss. Persistent storage is yet to be implemented for greater robustness.

4. Experiments/Evaluation/Tests

In the evaluation of the performance and scalability of our system, we made use of this dataset found on Kaggle, a dataset containing many URLs which is used in the testing scenarios. The dataset was filtered to obtain the safe URLs.

Our experiments are designed to investigate the retrieval capability of the system. Given a ShortURL, the metrics that we are interested in is the average number of RPC calls made, and the average time taken to retrieve a LongURL from the system.

To understand the scalability of the Chord Protocol, due to the use of finger tables, the number of lookups in the ring should scale logarithmically. In addition, the use of a cache will also reduce the amount of time taken and calls made in retrieving a URL. We will be investigating the logarithmic relationship, as well as the impact of having a cache in retrieving a URL.

Upon starting the script, the inputs to the program are the number of nodes and URLs which will be used to initialize the ring setup. Following which, upon selecting the EXPERIMENT option, specify the number of URL retrievals to be made. The experiment is simulated, and the total amount of time taken, total number of RPC calls, and their averages for both situations of using and not using a cache to retrieve the URLs are returned.

To validate correctness of the system, the total number of successful retrievals from running the experiments are also returned. The system should report that all the URL retrievals were successful.

To ensure fairness and consistency across different experiments as much as possible, they were run on the same laptop.

4.1 Experiment 1

In this experiment, we investigate how the number of nodes in the Chord Ring impacts the number of RPC calls made, and the time taken to find the node responsible for storing the ShortURL-LongURL pair in its URLMap.

The independent variable is the number of nodes in the Chord Ring, and we fix the number of URLs at 1000 and retrieval requests made at 2000. The results are presented in the table below, and the accompanying graphs (Figure A3 and A4) can be found in the Appendix.

	No cache		Cache		Difference	
Number of nodes	Average time (ms)	Average number of calls	Average time (ms)	Average number of calls	Average time (ms)	Average number of calls
5	4.198177	3.0165	3.114864	2.063	1.083313	0.9535
10	4.612834	3.264	3.932333	2.719	0.680501	0.545
20	7.282058	3.9285	6.503199	3.5195	0.409	0.409
100	49.04187	5.08	47.477797	4.987	1.564073	0.093
200	111.63104	5.708	110.95502	5.688	0.676016	0.02

Table 2: Statistics of Experiment 1 with varying number of nodes

From this experiment, it is clear that it demonstrates the scalability of Chord in terms of the average number of RPC calls made. When the number of nodes increases, the average number of RPC calls made is scaled in a logarithmic manner and not linearly. For the average time taken, there is a general increase in the average time taken to retrieve a URL.

When the cache is used, the metrics in general are lower than if no cache was used. This demonstrates the effectiveness of a cache. In addition, all experiments showed that all the URL retrieval requests were successful.

For the average time, we hypothesize that this could be due to performance issues. As timing is quite an arbitrary metric whereby the speed of execution is very dependent on the available CPU resources allocated to running the program, concurrent execution of other programs could potentially affect the timing results.

4.2 Experiment 2

In this experiment, we investigate the effectiveness of the cache at reducing the average time and average number of RPC calls made since the cache is hit more often when retrievals are repeated for the same URL.

For instance, if the number of URLs stored is 100, each URL will be requested on average 20 times, and if the number of URLs stored is 2000, each URL will be requested on average 1 time. This simulation is also representative of a real life network, whereby requests over a local subnet may be repeated quite often. The cache should store these requests so they can be served if it is queried again, and there is no need to query the system.

The independent variable is the number of URLs stored, and we fix the number of nodes in the Chord Ring at 50 and retrieval requests made at 2000. The results are presented in the table below, and the accompanying graphs (Figure A5 and A6) can be found in the Appendix.

	No cache		Cache		Difference	
Number of URLs stored	Average time (ms)	Average number of calls	Average time (ms)	Average number of calls	Average time (ms)	Average number of calls
10	25.927316	4.641	21.708259	3.852	4.219057	0.789
50	28.39442	4.6192	26.351701	4.3595	2.042719	0.2597
100	25.304555	4.799	23.505735	4.5585	1.79882	0.2405
500	26.951039	4.667	26.000409	4.511	0.95063	0.156
2000	25.782663	4.613	24.836577	4.508	0.946086	0.105

Table 3: Statistics of Experiment 2 with varying number of repeat requests

From this experiment, it is clear that it demonstrates the effectiveness of a cache when there are more repeat calls of a URL on average. The difference of the average time and average calls metrics for cache and no cache situation decreases as the average number of repeat calls made decreases. In addition, all experiments showed that all the URL retrieval requests were successful.

This is similar to real-world systems such as the Domain Name System (DNS), whereby requests made to frequently visited website domains are cached at various DNS servers to reduce latency and improve response times for subsequent requests. This minimizes the need for repeated requests to authoritative DNS servers. In this regard, the effectiveness of the cache is much greater when there are more repeated requests and is demonstrated in this experiment.

5. Concluding Remarks

The scalability and usefulness of a distributed system based on consistent hashing are demonstrated by the Chord-based URL shortening service. The system guarantees reliable and effective operation by incorporating features like successor lists, caching, consistent hashing, and fault-handling algorithms for both voluntary and involuntary node exits.

This project provided a valuable opportunity to bridge theoretical concepts with real-world implementation. While the core functionality of the system has been successfully demonstrated, challenges remain in areas such as fault recovery, cache design, and achieving a balance between stabilization speed and communication overhead. For instance, the stabilization rate (the rate at which Maintain() is called), set at 5ms throughout the project, allowed for rapid convergence but could benefit from further optimization to reduce bandwidth usage. Additionally, while the current prototype emphasizes design and functionality, it highlights the complexities of preparing a system for real-world deployment, especially in terms of scalability and robustness.

The project has also underlined the importance of observability and fault tolerance in distributed systems. These reflections provide a foundation for identifying key areas of improvement, detailed in the subsequent **Future Work** section, to elevate the prototype into a fully operational and scalable distributed system.

6. Future Work

To enhance the system's robustness, scalability, and overall functionality, the following improvements are proposed:

1. **Load Balancing Algorithm:** Implement a round-robin approach for better load distribution among Chord nodes, replacing the current random querying mechanism.
2. **Dockerization:** Refactor the implementation to run each Chord node in its own Docker container, providing better isolation and a consistent runtime environment. A detailed architecture for this is included in the appendices.
3. **Serving from Successor List:** Technically, the successor list in its current state can be used to serve requests; a simple few lines of code would enable this. However, time constraints prevented us from fully bug testing it, so it was not included in the final submission.
4. **Container Orchestration:** Use Kubernetes to manage Dockerized nodes, ensuring a scalable and reliable deployment for real-world applications.
5. **Monitoring and Logging Tools:** Integrate tools like Prometheus for capturing system metrics and Jaeger for distributed tracing, improving observability and debugging capabilities.
6. **Bootstrapping Service:** Introduce a mechanism for seamless discovery of existing nodes in the Chord ring, allowing new nodes in a Dockerized environment to join the system easily.
7. **Persistent Storage:** Augment the system to store URL mappings in both memory and persistent storage, improving data recoverability during node failures or restarts.
8. **Enhanced Visualization:** Develop a graphical user interface (GUI) for better visualization and monitoring of the Chord system.
9. **Retry and Timeout Logic:** Add robust retry and timeout mechanisms to enhance reliability under unstable network conditions.
10. **Cache Optimization:** Upgrade the current timestamp-based cache mechanism with a more efficient least recently used (LRU) policy.
11. **Stabilization Rate Tuning:** Experiment with stabilization rate adjustments to optimize the trade-off between convergence speed and communication overhead.

7. Appendix

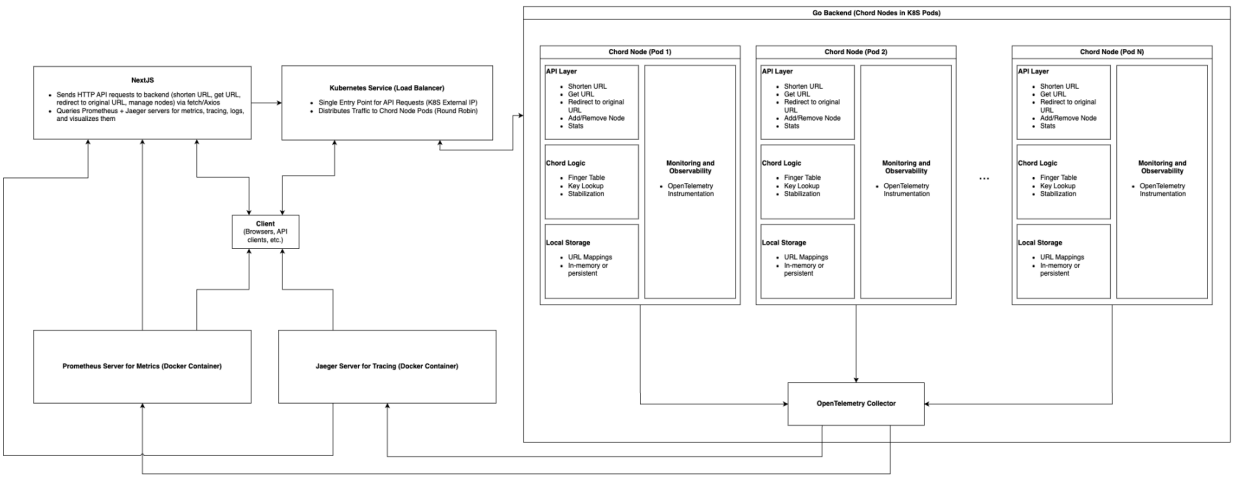

Figure A1: Potential architecture for a Docker + Kubernetes implementation.

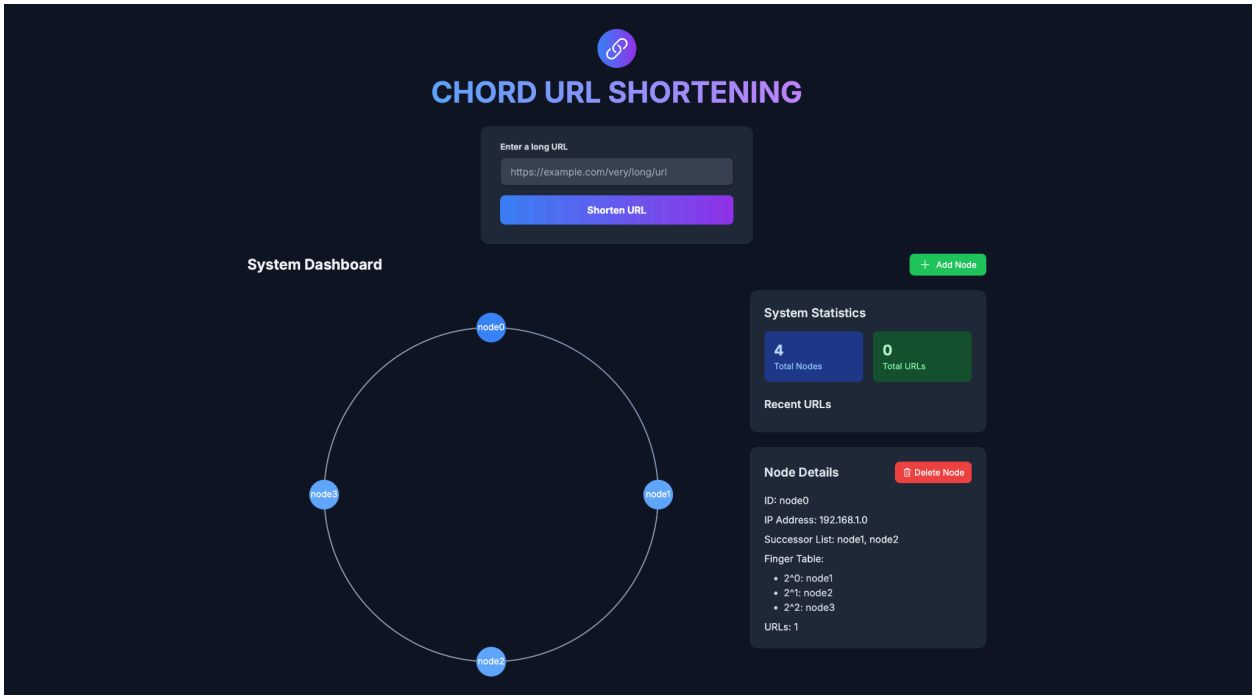

Figure A2: Possible GUI to better visualize and monitor the Chord nodes.

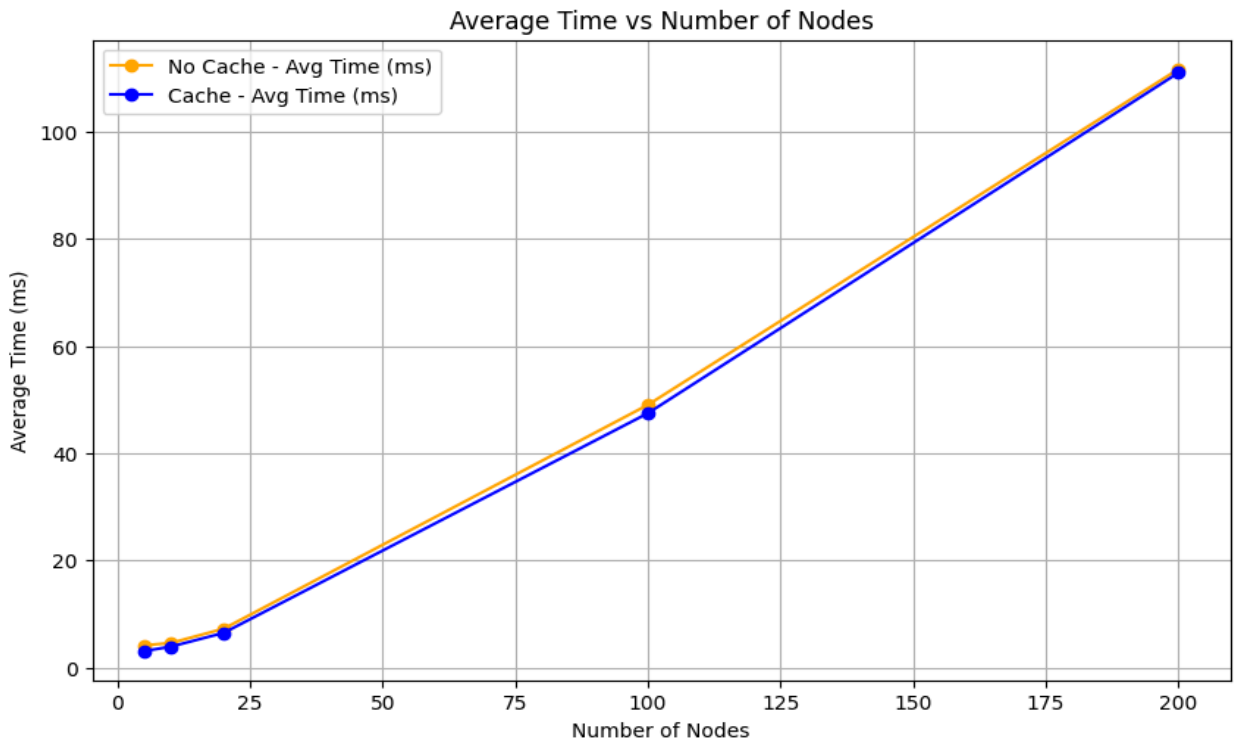

Figure A3: Experiment 1 Average Time(ms) vs Number of nodes

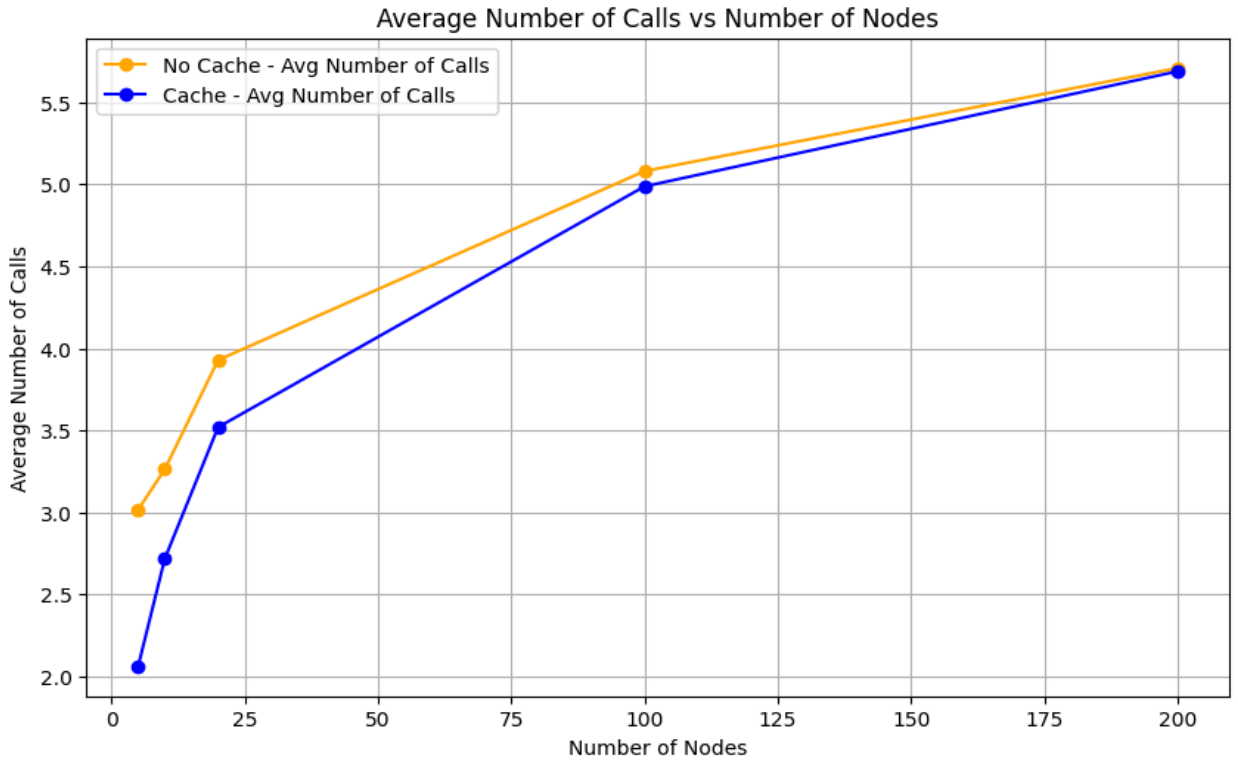

Figure A4: Experiment 1 Average number of RPC calls vs Number of nodes

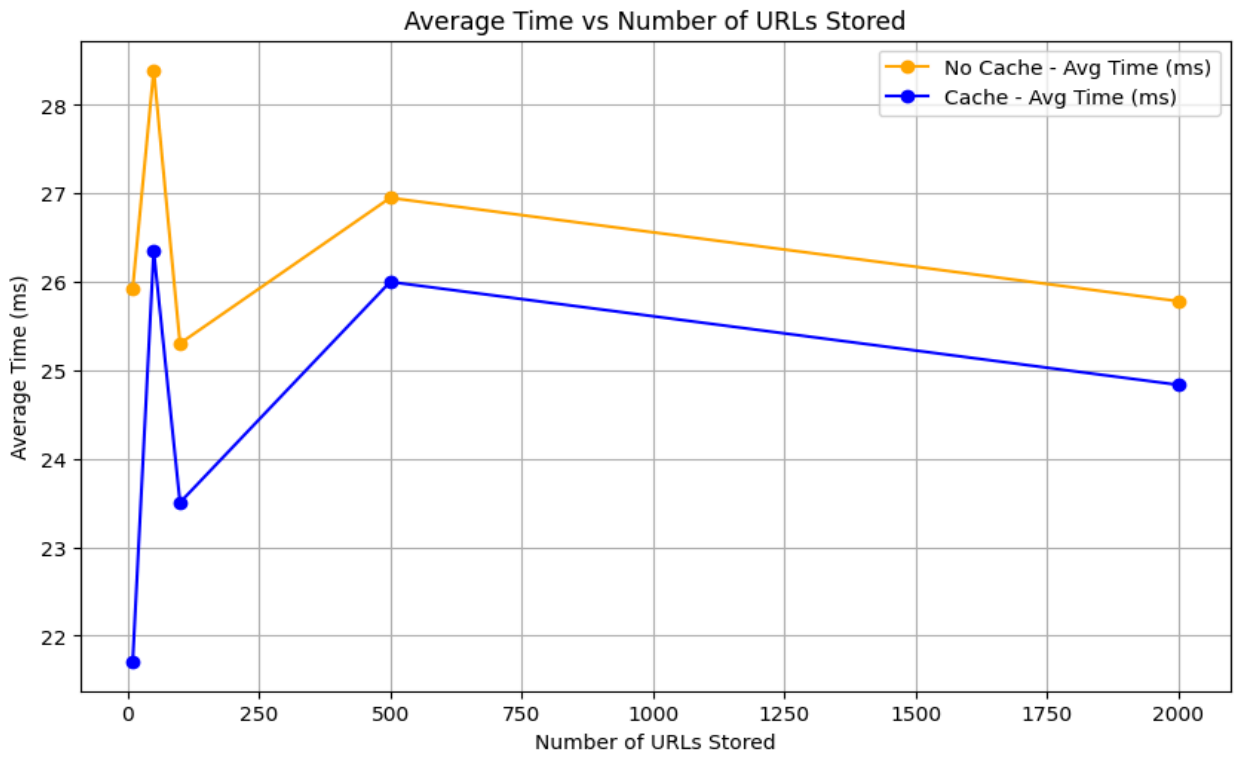

Figure A5: Experiment 2 Average Time(ms) vs Number of nodes

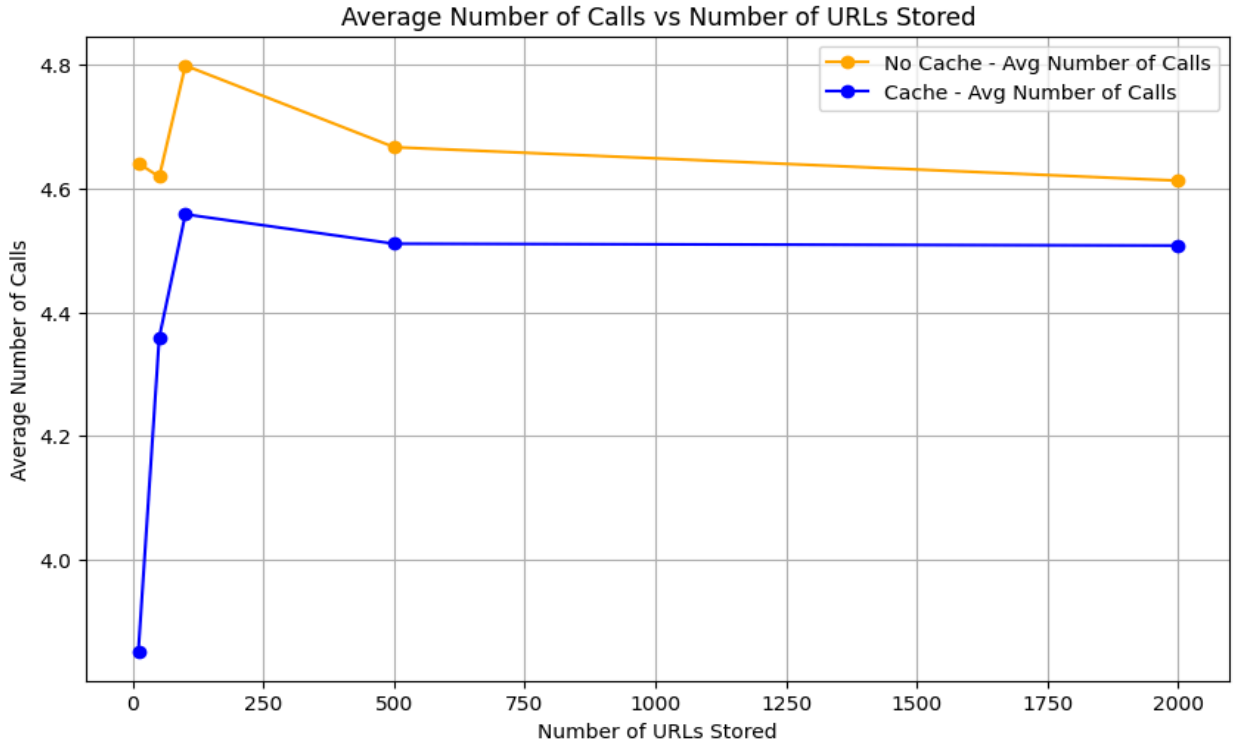

Figure A6: Experiment 2 Average number of RPC calls vs Number of nodes