ADS Assignment 2: Chord
Chord Distributed Hash Table

Bastien TALEC
Course: Advanced Distributed Systems (ADS)

October 4, 2024

Contents

1 Introduction

In this assignment, we investigate how scalable, resilient, and self-managing software systems can be implemented using peer-to-peer techniques, particularly the Chord Distributed Hash Table (DHT). The objective is to explore the mechanics behind structured distributed systems and demonstrate their scalability within a wide-area network environment. Specifically, we will implement the Chord DHT algorithms and utilize them in a distributed index service for message-based communication.

2 Implementation of the Chord DHT

This section describes the core algorithms and how they are implemented. The algorithms used include the finger table setup, lookup functionality, and other supporting mechanisms.

2.1 System Design

The system is designed around a service interface called `Messenger`, which allows developers to manage messages using a topic-based subscription model. The key functionalities include storing, retrieving, and deleting messages.

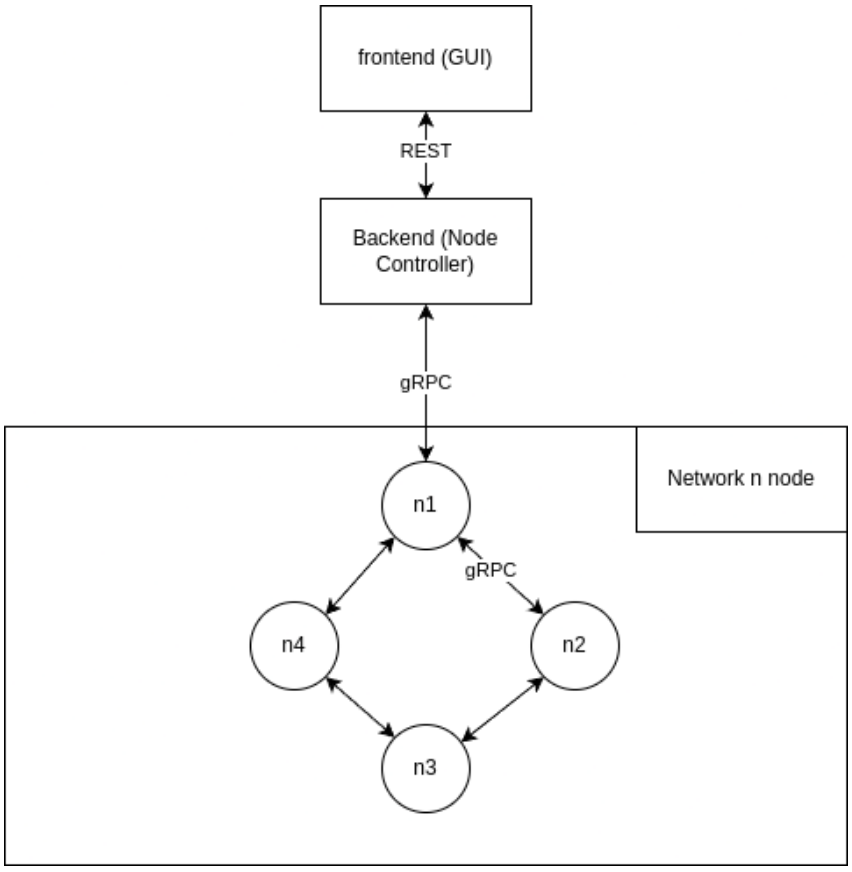

Figure 1: System Architecture of the Chord Message Management System

There are two services that use different protocols to manage the distribution of messenger data:

- `Node Network:` Each node is a Java program that runs on a different IP address and port and implement chord algorithm outlined in Section 4 of the Chord paper. Each node in the Chord network is a Java program that implements the Chord algorithm, as outlined in Section 4 of the Chord paper. Nodes are launched via the 'Main' class from the 'org.example' package, which starts a gRPC server and joins the Chord network.

  The 'Main' of class node accepts several command-line arguments for configuring the node:

  - `-host`: The host (IP address or domain) on which the node will run. The default is `localhost`.
  - `-port`: The port number for the node's gRPC server. The default is `8000`.
  - `-joinIp`: The IP address of an existing node in the network to join (bootstrap host).
  - `-joinPort`: The port number of the existing node to join (bootstrap port).
  - `-multiThreading`: Enables multi-threading for the node.

- `Backend:` The backend acts as a controller for adding or removing nodes from the network. It connects the GUI with the Chord network, allowing users to send and retrieve messages through its API.

- `Frontend:` A basic graphical interface to interact with and initialize the network. You can view information about each node in the network (such as the finger table, predecessor, and successor). It is primarily designed for sending and retrieving messages, as well as running performance tests.

2.2 Distributed Hash Table (DHT)

The Chord system uses a Distributed Hash Table (DHT) to place the node in the network and manage messages across multiple nodes. Each message is stored on a node based on its key, which is hashed to determine the responsible node. When a message is stored, its key is hashed to find the corresponding node in the Chord ring. The lookup operation uses the finger table of each node to efficiently locate the node responsible for the hashed key. Messages can then be stored or retrieved from the appropriate node.

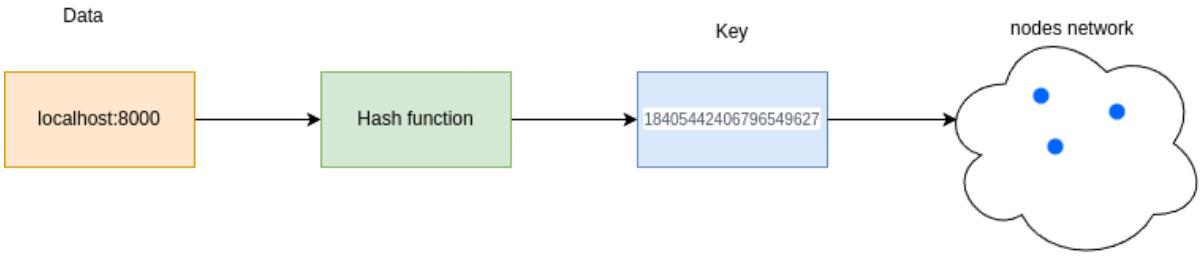

Figure 2: Distributed hash tables

Node ID Hashing Each node in the network is assigned a unique identifier (ID) based on its IP address and port. The node ID is generated using the SHA-1 hashing algorithm, ensuring a consistent distribution of nodes within the hash space. The following code illustrates how the node ID is generated by hashing the "ip:port" string, and then the result is taken modulo 2^{64} to fit within the Chord network's hash space:

```
// Function to hash the node ID based on its IP and port
private String hashNode(String input) {
    MessageDigest md = MessageDigest.getInstance("SHA-1");
    byte[] hashBytes = md.digest(input.getBytes());
    BigInteger hashInt = new BigInteger(1, hashBytes);
    BigInteger mod = BigInteger.valueOf(2).pow(64);
    return hashInt.mod(mod).toString();
}
```

By hashing the IP address and port, each node receives a unique position within the Chord ring, ensuring the even distribution of keys and minimizing collisions between nodes.

3 Evaluation

In this section, we evaluate the performance of the implemented system based on several parameters:

3.1 Number of Messages Scaling Evaluation

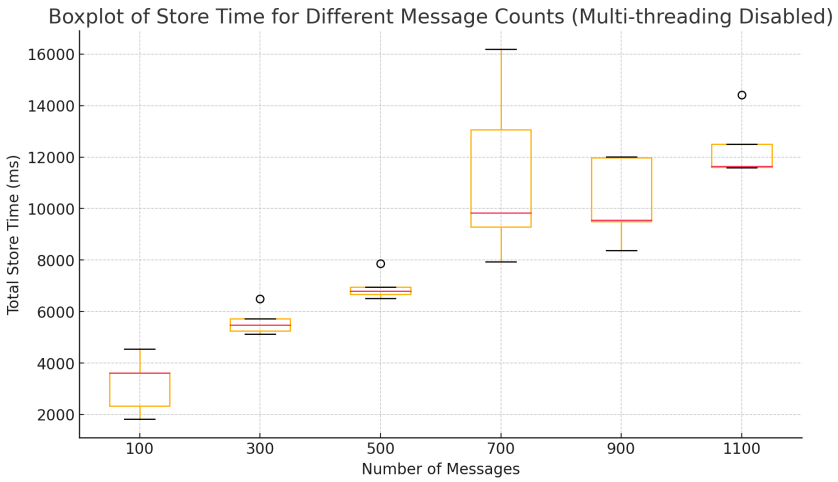

Figure 3: Boxplot of Store Time for Various Message Counts, Multi-threading Disabled

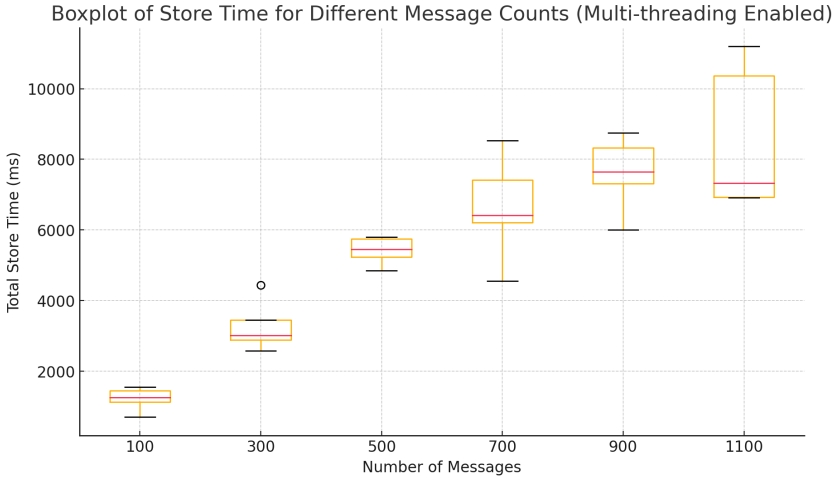

Figure 4: Boxplot of Store Time for Various Message Counts, Multi-threading Enabled

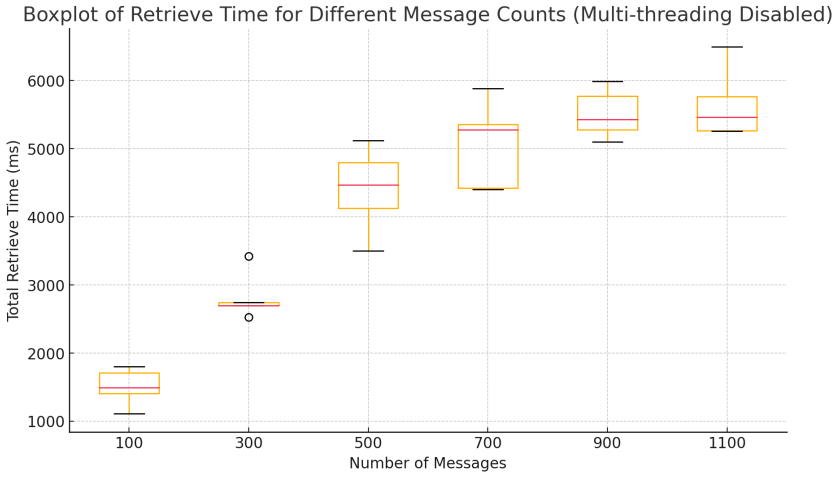

Figure 5: Boxplot of Retrieve Time for Various Message Counts, Multi-threading Disabled

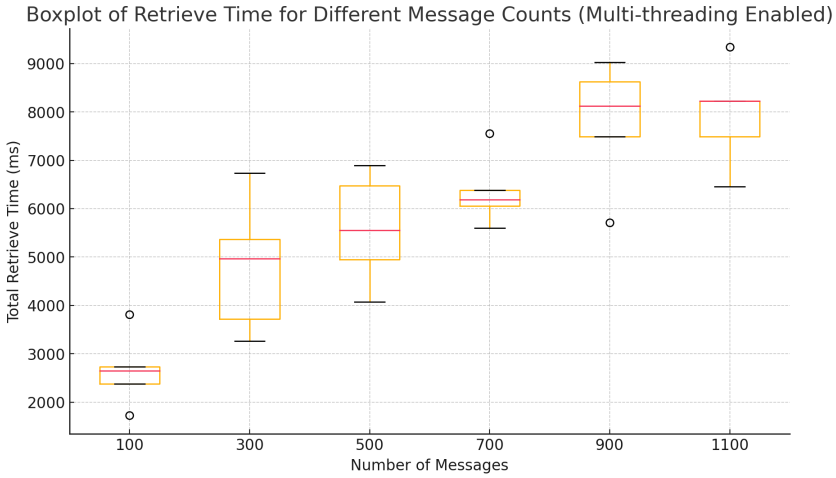

Figure 6: Boxplot of Retrieve Time for Various Message Counts, Multi-threading Enabled

- **Figure Description:** Each figure presents the performance results of five tests with increasing message counts (ranging from 100 to 1100 messages in increments of 200). The tests are conducted under two scenarios: with multi-threading disabled and enabled. Each test involved 5 nodes, with a data size of 1024 bytes. The boxplots illustrate the variation in store and retrieve times in milliseconds for each message count.

- **Interpretation:** The results show that as the number of messages increases, both store and retrieve times tend to increase. When multi-threading is disabled, the variance in times is generally higher, especially for higher message counts, as seen in the extended range of the boxplots. Multi-threading reduces both the average execution time and the variance, particularly for the store operation where it significantly speeds up the process compared to the non-multi-threaded case.

- **Observation:** The results indicate that multi-threading is not correctly implemented for the retrieve operation, as it consistently underperforms compared to the single-threaded version. Therefore, for the purposes of evaluating the performance gains from multi-threading, the focus should be on the store operation, where multi-threading significantly improves performance. In the following measurements, the retrieve-related graphs will

not be included as they provide no meaningful insights into multi-threading performance improvements.

3.2 Message Size Scaling Evaluation

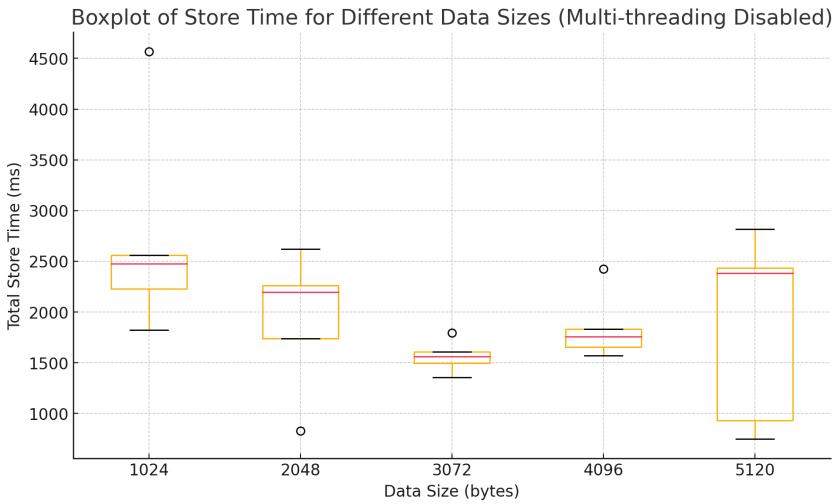

Figure 7: Boxplot of Store Time for Various Message Sizes, Multi-threading Disabled

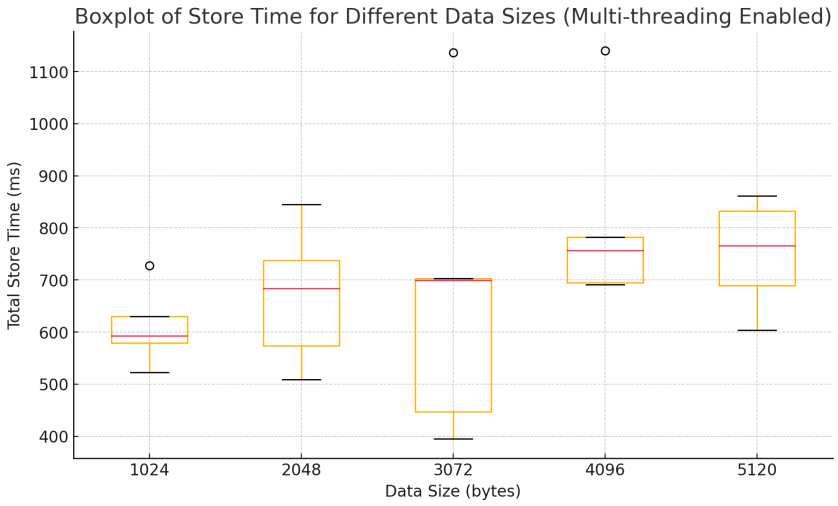

Figure 8: Boxplot of Store Time for Various Message Sizes, Multi-threading Enabled

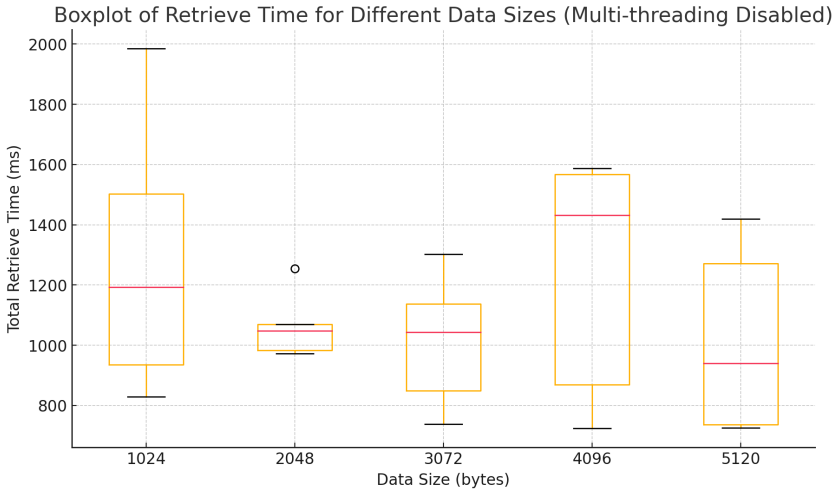

Figure 9: Boxplot of Retrieve Time for Various Message Sizes, Multi-threading Disabled

- **Figure Description:** Each figure presents the performance results of five tests with increasing message sizes (ranging from 1024 to 5120 bytes in increments of 1024). The tests are conducted under two scenarios: with multi-threading disabled and enabled. Each test involved 5 nodes, with 50 messages. The boxplots illustrate the variation in store and retrieve times in milliseconds for each message size.
- **Interpretation:** The message size appears to have a negligible effect on the overall performance compared to the communication time between nodes. The results show that the store operation is more sensitive to message size, where multi-threading provides noticeable performance improvements, particularly for larger messages. However, the communication overhead between nodes remains the primary factor impacting the total time.

3.3 Number of Nodes Scaling Evaluation

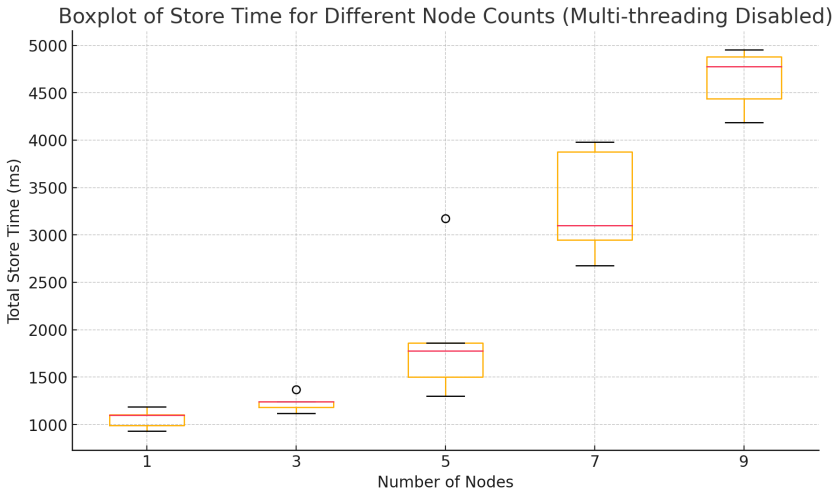

Figure 10: Boxplot of Store Time for Various Number of Nodes, Multi-threading Disabled

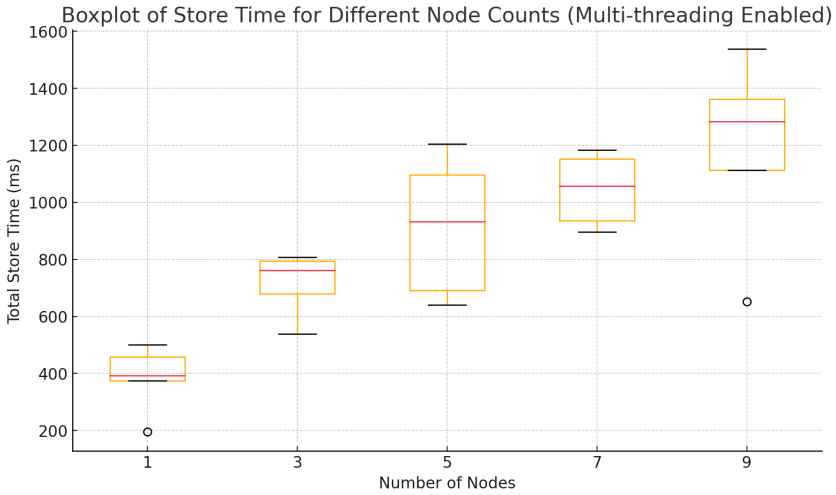

Figure 11: Boxplot of Store Time for Various Number of Nodes, Multi-threading Enabled

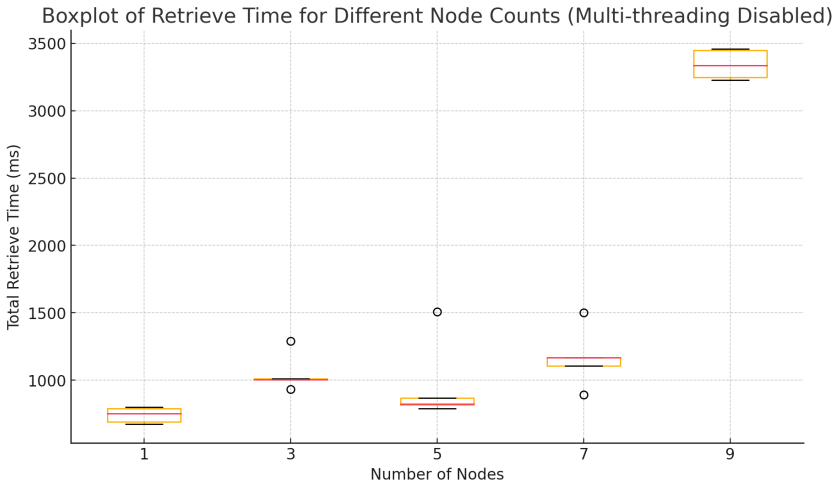

Figure 12: Boxplot of Retrieve Time for Various Number of Nodes, Multi-threading Disabled

- **Figure Description:** Each figure presents the performance results of five tests with increasing numbers of nodes (ranging from 1 to 9 in increments of 2). The tests are conducted under two scenarios: with multi-threading disabled and enabled. Each test involved 50 messages with a data size of 1024 bytes. The boxplots illustrate the variation in store and retrieve times in milliseconds for each number of nodes.
- **Interpretation:** The results show that the system's performance generally follows a logarithmic growth pattern (O(log n)) as the number of nodes increases. However, an anomaly was observed in the single-threaded retrieve test with 9 nodes, where the average retrieve time was abnormally high. This outlier may indicate a bottleneck in the system when scaling beyond a certain number of nodes, potentially due to network congestion or resource limitations.

3.4 Others metrics

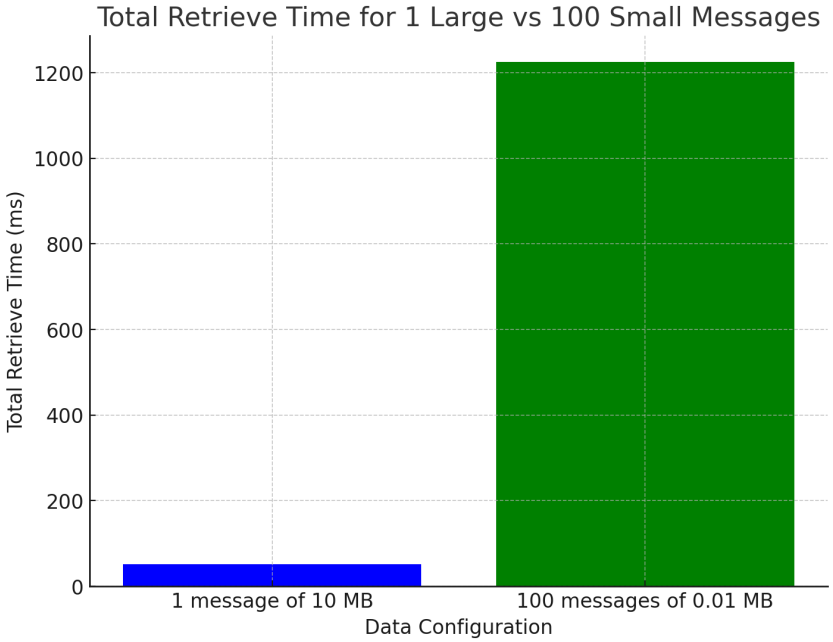

Figure 13: Retrieve time for Large message vs several small message

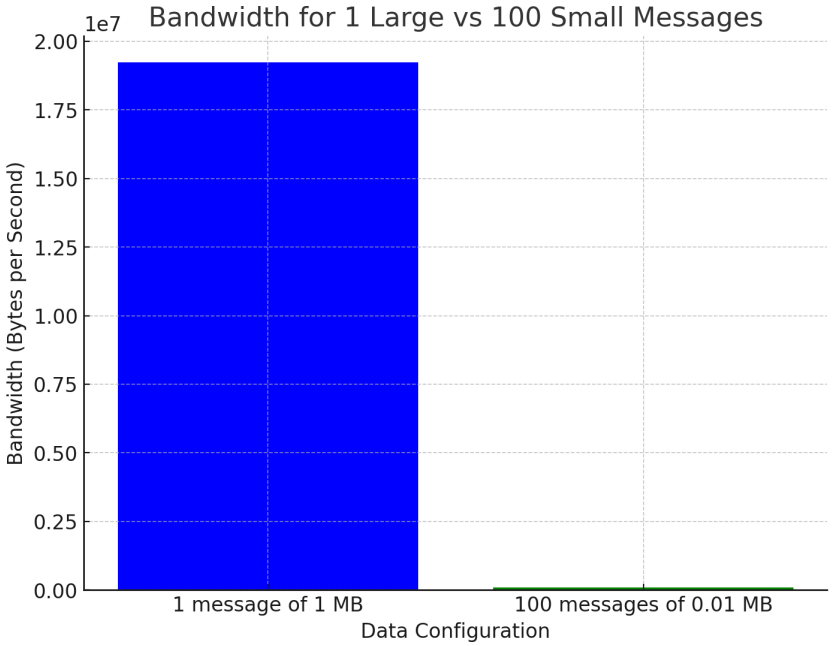

Figure 14: Retrieve bandwidth for Large message vs several small message

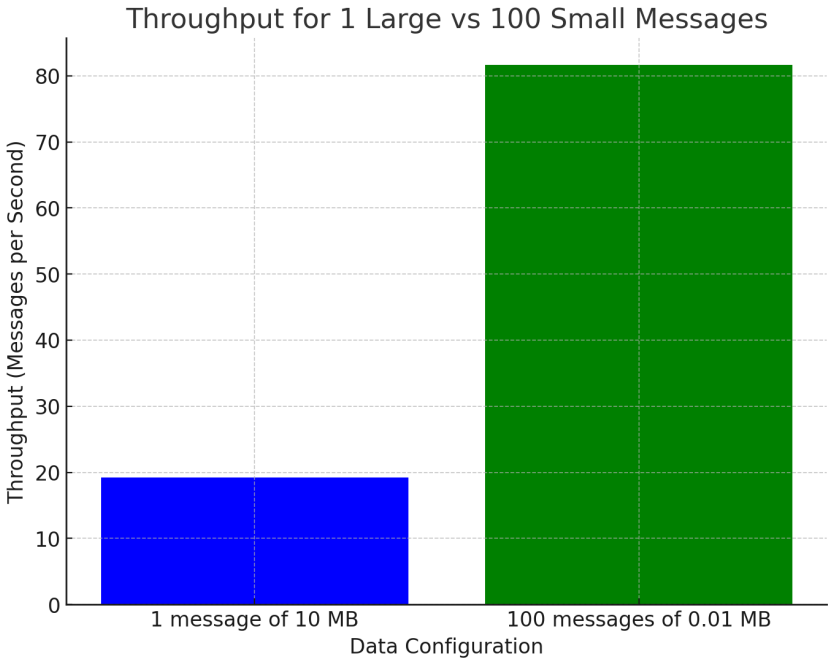

Figure 15: Retrieve throughput for Large message vs several small message

- **Figure Description:** Here are the bar charts comparing the total retrieve time, bandwith and throughput for one large message (1 message of 10 MB) versus multiple smaller messages (100 messages of 0.01 MB). The number of of node are fixed a 5.
- **Interpretation:** Retrieving many small messages increases total time due to communication overhead, while a single large message better utilizes bandwidth and reduces retrieval time. Communication is the most time-consuming factor, which is why P2P software like BitTorrent is optimized for downloading large files—it is more efficient to download one large file than many small ones.

4 Reliability of Node

In the Chord network, node reliability is an important aspect, particularly when nodes join or leave the network. In my implementation, when a node leaves the network, its predecessor and successor are updated to maintain the integrity of the ring. Specifically, when a node exits, the successor's predecessor is updated to the predecessor of the exiting node, and similarly, the predecessor's successor is updated to the exiting node's successor. This process ensures that the ring remains connected, even in the absence of the departing node.

Periodic Stabilization Functions The periodic functions such as `node.stabilize()` and `node.fixFingers()` play a critical role in maintaining the network's reliability. The `stabilize()` function ensures that each node correctly identifies its successor, allowing the network to self-correct in case of changes such as node departures or failures. The `fixFingers()` function periodically updates the finger table of each node, improving the efficiency of lookups by maintaining accurate routing information.

Limitations and Improvements Currently, in my implementation, when a node leaves the network, its stored messages are not transferred to other nodes. Additionally, there is no redundancy in message storage, meaning that if a node fails or leaves, the messages it was storing are lost. This is a limitation of the current design, and a possible area for improvement would be implementing redundancy mechanisms to replicate messages across multiple nodes to prevent data loss when nodes exit unexpectedly.

5 Conclusion

In this report, we implemented and evaluated a Chord-based Distributed Hash Table (DHT) system. The implementation demonstrated key features such as efficient message routing and node management using peer-to-peer techniques. Performance metrics such as throughput, bandwidth, and response time were analyzed through experiments with different message sizes and node configurations.

Our results show that, despite the poor implementation of multi-threading for message retrieval operations, we can anticipate similar performance gains as observed with the store operation. Optimizing this aspect could have allowed the deployment of more than 9 nodes by avoiding system limitations and bottlenecks. Addressing these inefficiencies would likely improve scalability and parallelization, ensuring better management of network connections.

Despite the need for further optimization of communication performance between nodes, our implementation still demonstrates the expected logarithmic complexity $O(\log n)$ for message retrieval as the number of nodes increases. This highlights the scalability of the Chord protocol even with the current performance limitations.

In terms of reliability, the network remains operational when nodes join or leave, but the lack of message redundancy leads to data loss if a node unexpectedly exits. Future improvements could focus on introducing redundancy mechanisms to enhance data reliability and fault tolerance in the network.

Overall, the Chord DHT system proves to be scalable and efficient for large data transfers, but optimizing communication between nodes and improving fault tolerance remain important areas for future work.