INF-3200: Distributed Systems Fundamentals

Assignment 1B: Distributed Hash Table based on the Chord protocol

Sera Madeleine Elstad and Victor Zimmer

October 2, 2024

I. Introduction

This report presents the implementation of a distributed key-value store inspired by the Chord protocol, a peer-to-peer system for efficient data lookup. The project aims to demonstrate scalability and performance in handling GET and PUT operations across multiple nodes in a decentralized network. Throughput was measured to evaluate system performance.

II. The chord protocol

The **Chord protocol** is a distributed lookup system designed for peer-to-peer networks. It provides efficient key-to-node mapping using **consistent hashing**, enabling scalable and fault-tolerant data storage in decentralized environments. Chord operates on a **circular identifier space** where both nodes and data keys are assigned hashed identifiers placed on a ring. Each node is responsible for the keys between itself and its immediate predecessor. By leveraging **finger tables**, each node stores information about a subset of other nodes, reducing lookup time to O(log N) [1].

Chord has multiple benefits. It scales efficiently in large networks, as each node maintains only limited routing information while still ensuring accurate lookups. The use of successor lists enhances fault tolerance by allowing continued operation despite node failures. Additionally, the decentralized nature of Chord eliminates single points of failure, and consistent hashing ensures a relatively even distribution of data across nodes, minimizing load imbalances [1].

However, there are some challenges. Chord fails to account for physical network layout, which might cause latency. Frequent node changes might temporarily destabilize the system, and finger tables increase storage overhead. Chord also lacks built-in security, leaving it open to specific attacks. Despite these problems, it is still an effective tool for distributed systems, providing both scalability and decentralization [1].

III. Design

In a Chord network, node IDs determine the placement of nodes in a circular identifier space, which is critical for efficient lookup and load balancing. Each key is mapped using consistent hashing based on the SHA-1 algorithm, which generates 160-bit hashes. For practical purposes, the identifier space for node placement of keys is restricted to 2^16 possible values, ensuring that the network can handle up to 65,536 participating nodes, the implementation could trivially be extended to 2^32 or 2^64.

A. Node ID assignment

Node IDs can be assigned using either hashed IDs or generated IDs.

Hashed IDs are created by hashing a node's unique identifying information, such as its hostname and port. While this method aims to distribute nodes randomly, it can lead to uneven spacing in small networks due to clustering effects, as shown in Figure 1. Hash collisions can also occur, where two nodes are assigned the same ID, disrupting the balance of the network. A uniform hash will eventually result in a uniform distribution of nodes over a large enough number of nodes.

To avoid these issues in smaller networks (e.g., 1, 4, 8, and 16 nodes, as illustrated in Figure 1), this implementation uses generated IDs. Generated IDs ensure even spacing by assigning nodes at predefined intervals, preventing clustering and hash collisions. Although generated IDs introduce some overhead, they result in better load balancing and consistent performance, particularly in smaller networks. This implementation places new nodes in the network by splitting the range of the node that is currently responsible for the longest range.

B. Finger table, successor and predecessor

Each node tracks its successor and predecessor, and can forward requests to them if its not within its range of responsibility. By forwarding requests we can expect a time complexity of O(N)

Finger tables are used to increase routing efficiency by decreasing lookup times to O(log N). Each node keeps a finger table with pointers to other nodes located around the ring. This allows nodes to efficiently send requests to the right node while limiting the number of required lookups, by routing to the node in the finger table that is closest to the location of the key.

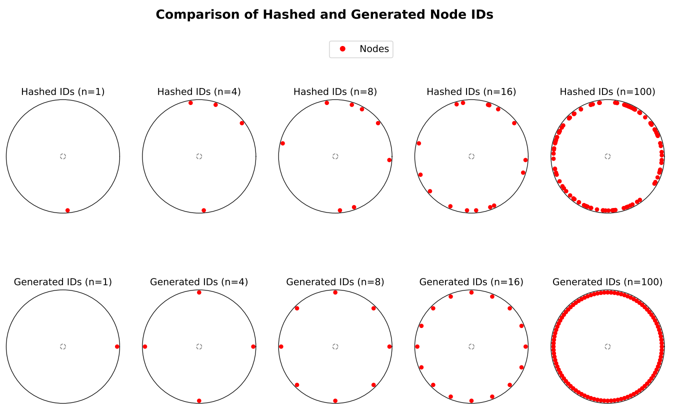


Fig. 1: Example of hashed vs generated node IDs

IV. Our implementation

This implementation of the Chord Distributed Hash Table (DHT) protocol is written in Rust, utilizing the Rocket web framework for HTTP request handling. The system operates as a set of nodes, each representing a server in the Chord ring. Each node handles requests to store and retrieve key-value pairs, as well as routing data to the appropriate node responsible for a specific key, according to the Chord protocol.

The Node structure is the implementation's central component, containing critical information for each network node. This provides information such as the hostname, port, position in the Chord ring, and the keys for which the node is responsible. The Node structure allows each node to be uniquely identified and efficiently located inside the Chord ring. Each node knows its immediate predecessor and successor. If there is only one node, it will be its own predecessor and successor.

The system state is controlled by this Node structure, which ensures thread-safe access across HTTP request handlers. This structure includes information about the local node, including its finger table, successor list, and predecessor, which is required for routing operations. The finger table is a critical component of the implementation, allowing logarithmic time lookups by storing pointers to nodes at exponentially increasing distances in the ring. This enables nodes to efficiently transmit requests to the intended destination.

A simple in-memory key-value store is used to store data, which is supported by a hash map and contained in a Storage object. The system performs basic key-value pair storage and retrieval activities. Keys are mapped to nodes using the SHA-1 hashing technique, which ensures consistency. This ensures that keys are distributed evenly and consistently across the ring, allowing for balanced load distribution throughout the network.

A. *Multithreading*

The Rocket.rs framework we use for handling HTTP routing automatically enables multithreading in our application, and we have not disabled this feature. As such, it has been designed to be thread-safe by utilizing the Rust concurrency primitives Arc (Atomic Reference Count) and RwLock (Read-Write Lock).

Arc is a smart pointer that allows for shared ownership of data between threads. It provides a way to safely share data between multiple threads by incrementing and decrementing a reference count, ensuring that the data is only accessed when there are active references to it.

RwLock, on the other hand, is a synchronization primitive that allows for both read-only and write access to shared data. When a thread attempts to write to the config, the RwLock will block any other threads that try to access the data simultaneously until the writer has completed its operation. This ensures that the data remains consistent and free from race conditions.

This design provides significant benefits in terms of scalability, reliability, and development ease. By allowing multiple threads to read data concurrently, our application can handle a larger volume of requests without incurring significant performance overhead. The use of RwLock ensures that writes are atomic and consistent, reducing the risk of data corruption or inconsistencies. Additionally, Rust's built-in concurrency primitives abstract away low-level concurrency concerns.

This design allows for unlimited simultaneous readers, but ensures that if multiple threads attempt to write to the nodes local config state at the same time, access will be blocked until one thread completes its operation. Overall, this approach enables our application to take full advantage of multiple threads and provides a robust foundation for building a scalable and reliable system.

Communication between nodes is handled via HTTP endpoints. Each node exposes a RESTful API with endpoints for GET and PUT operations. These operations allow clients to store values associated with keys and retrieve values by key. The system also includes endpoints to manage node joining and leaving.

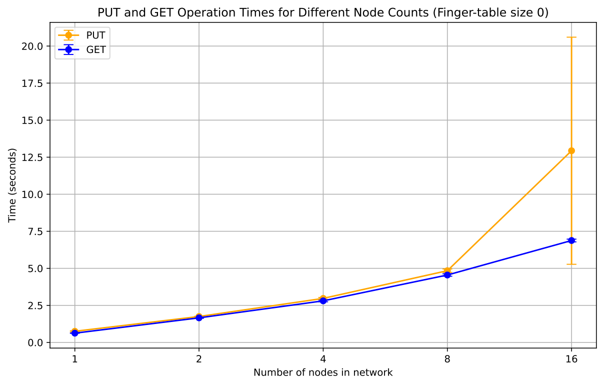
V. Experiment

Two experiments were conducted to evaluate the performance of the system. In the first experiment, the system was tested with networks of 1, 2, 4, 8, and 16 nodes without using finger tables. The goal was to measure the impact of increasing network size on the time taken for PUT and GET operations when lookups are done through a linear search. In the second experiment, the system was tested with a fixed network of 16 nodes, varying the size of the finger table (0, 2, 4, 6, and 8) to evaluate how finger tables affect lookup performance.

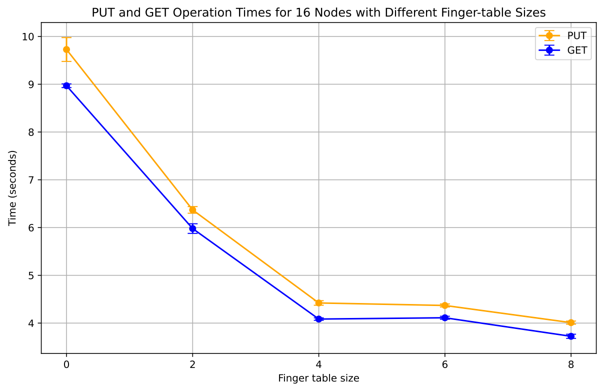
For each network configuration, a series of PUT and GET operations were performed, and the time taken to complete these operations was measured. Each experiment was repeated three times to ensure reliable results, and the average time for each operation was calculated along with the standard deviation. The results for the first experiment, showing the time required for PUT and GET operations without finger tables, are presented in Figure **??**, while the results for the second experiment, showing the impact of varying finger table sizes on a 16-node network, are displayed in Figure **??**.

PUT and GET Operations Without Finger Tables:
In the first experiment, where no finger tables were used, PUT and GET times increased significantly as the number of nodes grew. For example, with 1 node, PUT operations took an average of 0.75 seconds and GET operations 0.63 seconds. However, with 16 nodes, PUT operations took 12.94 seconds, while GET operations took 6.88 seconds. Notably, the error bars for the PUT times on 16 nodes were particularly large, suggesting significant variability between runs. This variability may be due to inconsistencies in network communication or bottlenecks when many nodes are involved in the PUT operations, causing delays.

Effect of Finger Tables on 16 Nodes:
In the second experiment, using finger tables significantly improved the performance of the 16-node network. Without finger tables (size 0), PUT operations took 9.73 seconds and GET operations 8.97 seconds. As the finger table size increased, the operation times decreased, confirming the logarithmic reduction in lookup complexity. With a finger table size of 8, PUT operations took 4.01 seconds and GET operations took 3.72 seconds, demonstrating that finger tables improves routing.



(a) PUT and GET for different nodes



(b) PUT and GET for 16 with different finger table size

Fig. 2: PUT and GET for nodes, with and without finger table

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.