Distributed Hash Table

INF3200 Assignment 2

Tord Fylkesnes Nytun
tornyt1573@uit.no

I. Introduction

A distributed system can be defined as a collection of independent computers that appears to its users as a single coherent system [1]. Structured peer-to-peer overlays, such as Chord, are a prominent example of distributed systems designed to achieve scalable lookup services [1].

Distributed systems increasingly rely on decentralized mechanisms for storing and retrieving data efficiently. Traditional client-server approaches have scalability and fault tolerance limitations, particularly when the number of clients grows or when servers fail. Distributed Hash Tables (DHTs) offer a decentralized solution by spreading data and responsibility across multiple peers in a structured manner. Among DHT protocols, Chord [2] is one of the most influential due to its elegant design, provable scalability, and robustness against churn (rate at which nodes join and leave the network).

This report documents the design, implementation, and evaluation of a Chord-based DHT written in Go. The implementation focuses on correctness of key placement, lookup efficiency, and resilience against identifier collisions. The experimental evaluation measures throughput of the system under different network sizes, using PUT and GET operations as the performance metric. The results are presented with error bars to highlight performance variability across multiple runs.

The report is structured as follows: Section II presents the technical background of Chord. Section III details the design and implementation decisions, including issues encountered and how they were resolved. Section IV describes the experimental methodology and presents the results. Section V discusses the outcomes, their implications, and limitations of the system. Section VI concludes with lessons learned and possible improvements.

II. Technical Background

A. Distributed Hash Tables

A Distributed Hash Table is a decentralized data structure that allows peers in a network to store and retrieve key-value pairs without a central authority. Each peer is assigned an identifier from a large identifier space, and responsibility for keys is distributed across the peers. By ensuring that responsibility is deterministically defined, a DHT guarantees that any key can be located in a bounded number of steps.

B. The Chord Protocol

Chord maps both nodes and keys to an m-bit identifier space using consistent hashing. Identifiers are arranged on a logical ring modulo 2^m. Each node is responsible for the interval of keys between its predecessor and itself. Lookups are resolved by routing queries clockwise along the ring until the responsible node is reached.

Chord achieves efficiency by introducing the *finger table*, a routing structure of m entries per node. The ith entry in the finger table points to the first node that succeeds the current node by at least 2^{i-1} on the ring. This allows lookups to be resolved in $O(\log N)$ hops on average, where N is the number of participating nodes.

C. Stabilization and Fault Tolerance

Because nodes may join or leave at any time, the system must adjust to maintain correctness. The Chord protocol includes background tasks for stabilization, predecessor checks, and finger table fixes. These ensure that routing remains correct despite churn. Each node only stores a logarithmic amount of state, making the protocol highly scalable. The joining, leaving and stabilization protocols are not implemented in this project.

III. Design & Implementation

A. System Overview

The system was implemented in Go, chosen for its concurrency primitives and networking support. Each node runs as a separate process and communicates over RPC-like calls. Keys are generated by hashing input strings using SHA-1 and truncating to an m-bit identifier. Initially, $m = 8$ was chosen due to low node count and simplicity, resulting in an identifier space of $2^8 = 256$. However, this small space quickly led to frequent collisions in node identifiers, which in turn caused infinite lookup loops. To resolve this, $m = 16$ was adopted, expanding the identifier space to $65,536$ and greatly reducing collision probability.

Consistent hashing is a standard mechanism for mapping keys to nodes in a scalable and load-balanced manner [1]. To minimize identifier collisions, it is common to increase the size of the identifier space, as we did by choosing $m = 16$ [1].

B. Node State

Each node maintains the following state:

- Its identifier in the 2^m space.
- Identifier and address of its predecessor.
- Identifier and address of its successor.
- A finger table of size m.
- A local thread-safe key-value map containing the keys for which it is responsible.

C. Interval Checks

Correct interval checks are crucial for determining key ownership and routing. The implementation distinguishes between open, left-inclusive, and right-inclusive intervals, as required by the protocol. In particular, a node n is responsible for a key k if $k \in (n.\text{predecessor}, n]$. Errors arose in earlier implementations because checks were made with the wrong boundaries.

D. Concurrency and Communication

Nodes communicate using gRPC. Each request is handled in a goroutine, allowing concurrent servicing of multiple operations. Synchronization was required to ensure thread safety of the local key-value store and finger table updates.

E. Failure Modes

The implementation encountered two main failure modes:

1) **Node identifier collisions:** Solved by expanding m from 8 to 16.
2) **Infinite loops in lookup:** Solved by correcting interval checks and ensuring that the closest preceding finger is selected correctly.

IV. Experiments

A. Methodology

The experiment evaluates throughput of the DHT under different network sizes: 1, 2, 4, 8, and 16 nodes. For each configuration, 1000 PUT operations and 1000 GET operations were performed. Each configuration was repeated for at least five trials to capture variance. New servers were started for each trial, each on a random port. The benchmark script selects a random node for each PUT/GET request. Throughput is measured as operations per second, computed by dividing the number of operations by the elapsed time. Results are aggregated to compute the mean and standard deviation, which are visualized with error bars.

A known limitation of the current implementation is that sometimes a server fails to start due to race condition between finding an available port and server starting with the respective port. This is most likely due to other user activity on the cluster.

B. Results

The results are shown in Figure 1, which presents the average throughput of PUT and GET operations across the tested network sizes. Error bars indicate standard deviation across multiple runs.

Throughput decreases as the number of nodes grows. With a single node, both PUT and GET operations achieve more than 500 operations per second. At two nodes, throughput drops into the 400–470 operations/s range, while four nodes yield approximately 250–350 operations/s. For eight nodes, throughput stabilizes around 200–250 operations/s, and with sixteen nodes it is typically in the range of 160–206 operations/s.

PUT and GET operations scale in a similar fashion across all network sizes, and the standard deviation is largest for mid-sized networks (around four nodes), with values fluctuating between approximately 247 and 367 operations/s. For very small (1–2 nodes) or very large (8–16 nodes) networks, variance is lower.

V. Discussion

The observed decrease in throughput with larger network sizes reflects the logarithmic lookup complexity of Chord and the associated communication overhead. Small networks (1–2 nodes) achieve very high throughput because most requests resolve locally or with minimal routing. Mid-sized networks (around four nodes) experience more variance, likely due to some requests traversing multiple hops while others remain local.

Increasing the identifier space from $m = 8$ to $m = 16$ was critical: smaller m caused node collisions and occasional infinite lookup loops, even with the maximum number of 16 nodes. With $m = 16$, requests were properly routed and errors avoided, confirming the need for a sufficiently large identifier space to avoid collisions.

PUT and GET throughput remain close because both operations require similar lookups to locate the responsible node. The system scales roughly according to $O(\log N)$ lookups per operation, as expected from Chord theory [1].

Overall, the results confirm that our implementation behaves as expected: throughput declines gradually with network size, PUT and GET rates remain similar, and variance is largest for intermediate network sizes due to the interplay of routing, latency, and load distribution.

VI. Conclusion

This report presented the design, implementation, and evaluation of a Chord-based Distributed Hash Table. The implementation encountered challenges with identifier collisions and interval checks, both of which were resolved. The experimental evaluation showed that throughput decreases as the number of nodes increases, consistent with Chord's logarithmic lookup complexity. However, the system maintained reasonable performance, with throughput greater than 170 operations per second at 16 nodes.

Future improvements could include:

- More sophisticated load balancing to handle skewed key distributions.
- Fault injection experiments to evaluate robustness under churn.
- Scaling experiments with larger numbers of nodes.
- Support for dynamic leaving and joining of nodes.

Overall, the implementation confirms the scalability and robustness of the Chord protocol, while highlighting practical issues that arise in real deployments.

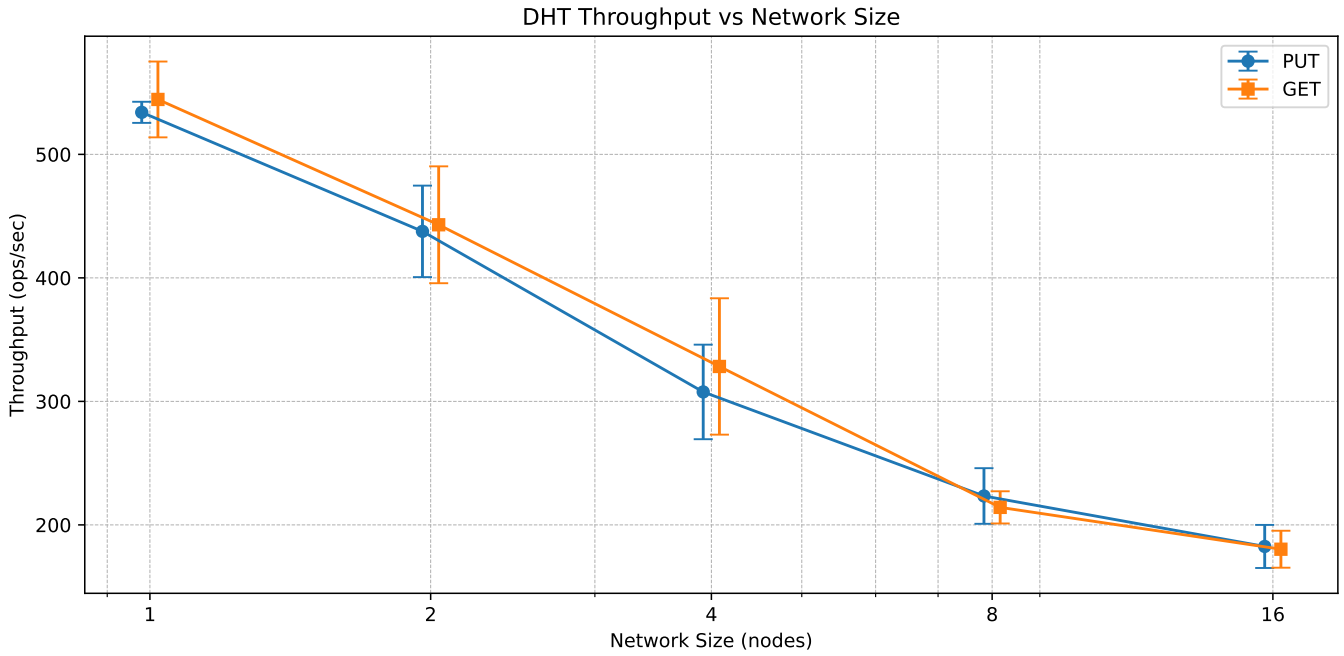

Fig. 1. Throughput as a function of network size. Error bars show standard deviation across trials.

REFERENCES

[1] M. van Steen and A. S. Tanenbaum, *Distributed Systems*. Maarten van Steen, 2023.

[2] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. 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.