INF-3200: Distributed System Fundamentals

Assignment 2: Distributed Hash Table and the Chord Protocol

September 28th, 2025

August S. Hindenes
Department of Computer Science
UiT The Arctic University of Norway
Tromsø, Norway
ahi025@uit.no

Karoline W. Benjaminsen
Department of Computer Science
UiT The Arctic University of Norway
Tromsø, Norway
kbe173@uit.no

I. Introduction

This report outlines the solution for Assignment 2 in INF-3200. The objective of the assignment was to give us practical experience in designing and implementing a distributed system by developing a distributed hash table. The implementation was based on the Chord protocol, a peer-to-peer lookup protocol designed to efficiently identify the node responsible for a given key. Chord maps keys to nodes in a scalable way that ensures communication cost scales logarithmically with the number of nodes [1]. To demonstrate this property in practice, we conducted experiments measuring throughput in terms of `GET` and `PUT` operations, thereby evaluating and validating the system's performance and scalability.

The rest of this report is organised as follows. Section II outlines concepts and background information relevant to the assignment. Section III describes the general design and mechanism of the Chord protocol, while Section IV describes the specifics of our solution and its implementation. Section V describes our methodology for testing and measuring our solution. Section VI gives the results of our experiment. Section VII discusses the solution and results. And finally, Section VIII is the conclusion.

II. Technical Background

A *distributed system* consists of multiple interconnected computers that share processes and resources to enhance scalability, reliability, and overall efficiency [2]. *Peer-to-peer* (P2P) systems are a type of distributed system characterised by the absence of centralised control or hierarchical organisation, with each node running equivalent software capable of performing the same functions [1].

The core operation and fundamental challenge in most peer-to-peer systems is efficiently locating and retrieving data items [1]. *Distributed hash tables* (DHTs) provide a structured solution to this challenge. Similar to a traditional hash table, a DHT stores data as key-value pairs, but allows any participating node in the network to locate and retrieve the value associated with a specific key. [3].

A. The Chord Protocol

A distributed hash table requires some method for dividing the key-value pairs among nodes and for directing requests to the node responsible for a particular key. *Chord* is a protocol designed for peer-to-peer distributed hash tables that defines how keys are assigned to nodes and how a node can locate the value associated with a given key by identifying the node responsible for it [4].

Chord assigns keys to nodes using *consistent hashing*, a technique that assigns each node and each key to a unique *identifier*. These identifiers are ordered in a an *identifier circle*, sometimes called a ring, which allows the system to evenly distribute keys across all participating nodes. Key location in a Chord network can be performed in two ways. The simplest approach is a simple linear lookup, where each node forwards a query to its immediate successor until the node responsible for the key is reached. While straightforward, this method is slow, requiring lookup time $O(N)$. To improve efficiency, Chord can employ a *finger table* at each node, which contains information about a subset of nodes around the identifier circle. By using the finger table, a node can forward a query to the closest preceding node to the target key, jumping across the network. This reduces lookup time to $O(logN)$ [2].

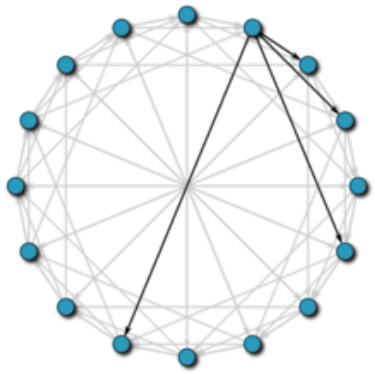

Fig. 1. A 16-node Chord network with a finger table [4]

There are multiple benefits to using Chord. Its decentralised architecture eliminates the need for a central authority, thereby avoiding potential performance bottlenecks and removing single points of failure. The use of consistent hashing ensures that keys are evenly distributed among nodes, providing balanced load among nodes, and preventing hotspots. Chord's finger table mechanism enables efficient lookups, reducing the number of routing hops. The logarithmic complexity allows the system to scale to a large number of nodes without an increase in lookup cost. Another strength of Chord lies in its simplicity. The protocol focuses on a single core operation: mapping a key to the node responsible for storing its value. This makes it relatively easy to implement in practice [1].

Nevertheless, Chord also has limitations. The protocol fails to account for physical network layouts. The lookups, although logarithmic in theory, may still require multiple hops across a wide-area network, resulting in higher latency in practice. The protocol only provides basic key-to-node mapping, which means that additional features such as caching and support for complex queries must be built on top of it, adding complexity to the overall system. Finally, Chord is susceptible to malicious attacks due to its reliance on finger table information, which could be falsified by malicious nodes [5].

III. Design

A. Node Assignment

We use an identifier ring of size 2^M with M=64. Each node is given a 64-bit identifier, calculated by taking the address ("host:port") which is then hashed using SHA-1 and masked to 64-bit. Considering the number of nodes used in this assignment it is not necessary to use an identifier ring this large, but due to the nature of Chord there are no downsides to using 64-bit over 32- or 16-bit. Roomier rings actually reduce the chance of node-ID collisions and thus allow for more growth of the cluster size without having to reconfigure. The limiting factor for the identifier ring is the maximum size of SHA-1 hashes (160-bit), but using identifiers larger than 64-bit, you could theoretically start seeing a slowdown due to overhead related to >64-bit operations in the CPU.

The same hashing function is used for the keys, resulting in key identifiers in the same identifier space as the nodes. We take a full static list of all nodes in the cluster (no dynamic join/leave) and hash the ID of each node. Then we sort the list clockwise (smallest to largest ID) and assign key responsibility using the open-closed interval rule $(\text{predecessor}(n), n]$. This gives a balanced key distribution through consistent hashing, and since the nodes are static, there is no balancing overhead.

B. Successor and Finger Table

After sorting the ring, each node's immediate *successor* and *predecessor* are the clockwise and counter-clockwise neighbours, respectively. For efficient routing, we build a size-m) finger table for each node. By default, the size of the table is $m = \lceil \log_2 N \rceil$ (configurable), and entry i (for offsets $2^{M-m} \dots 2^{M-1}$) stores the first node whose ID is $\geq (n + 2^i) \bmod 2^M$. When looking up a key k, we forward to the closest preceding finger of k. If none of the fingers precedes k we forward the request to the successor. This reproduces Chord's $O(\log N)$ hop behaviour on a static ring.

IV. Implementation

Our implementation is in Rust using an Actix-web service exposing `/storage/{key}` for `GET/PUT`. On initialization (`/storage-init`) the node receives the complete list of nodes in the cluster, it then computes 64-bit identifiers via SHA-1, sorts the ring, finds *predecessor* and *successor*, and builds the finger table. In the storage `GET/PUT` endpoints, responsibility is checked with the interval $(\text{predecessor}(n), n]$. Requests for which the current node is not responsible are forwarded using the *closest preceding finger* rule. To prevent infinite hop loops, we enforce a hard cap by adding a hop counter in the header of each forwarded request. Since dynamic join/leave is not supported, stabilization and background maintenance are unnecessary. `/reconfigure` rebuilds the ring and fingers from a new static list of nodes whilst also allowing a configurable finger table size to be set and reset the local hash table of the node. `/reconfigure` is used by the throughput test script to change the cluster and finger table size allowing. We have also implemented middleware that triggers automatic shutdown of the node if it is not used for x minutes.

V. Experiments

To evaluate the performance and scalability of our Chord implementation, we designed a series of experiments to test the throughput of our DHT under varying system configurations. Specifically, we investigated how two parameters, the number of nodes in the system and the size of the finger table (M), would affect performance.

A. Experimental Setup

Throughput was measured in terms of successful `PUT` and `GET` operations per second. Each test consisted of inserting 1,000 randomly generated key-value pairs into the DHT and then retrieving each key to verify correctness. To ensure consistency across runs, the keys were generated using UUIDs, and values were randomly composed of alphanumeric characters. The total execution time for all operations was measured, and the throughput was then calculated as the total number of operations divided by the total execution time.

Two parameters were varied during the experiments:

- Node counts: $\{1, 2, 4, 8, 16, 32\}$
- Finger table sizes (M): $\{0, 1, 2, 4, 8\}$

Testing with finger table size equal to 0 is equivalent to no finger table. For each configuration, the system was reconfigured to use the specified number of nodes and finger table size. Every test was repeated three times, reporting the average throughput along with the standard deviation to account for variability. Additionally, we measured the average time per operation as an alternative performance metric.

VI. Results

The throughput of the DHT implementation decreased as the number of nodes increased, but larger finger tables consistently improved performance at each scale, as shown in Figure 2. With a single node, throughput remained high across all configurations of finger tables, ranging from 1,530 to 1,566 operations per second. At 32 nodes, throughput fell to 72 ops/s for $M = 0$ and to 260 ops/s for $M = 8$.

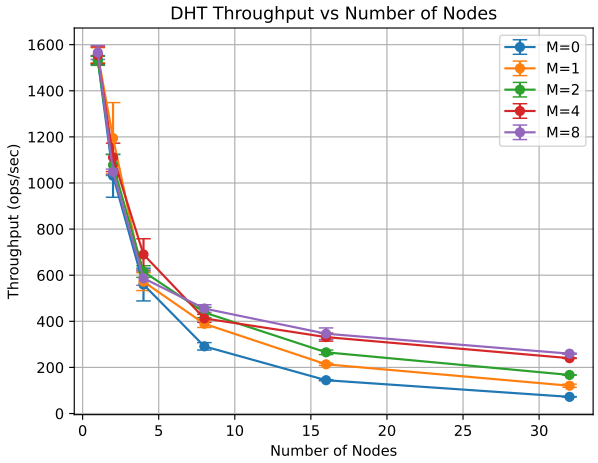

Fig. 2. The DHT throughput measured with an increase in the number of nodes and finger table sizes.

The average time per operation showed that operation times increased with the number of nodes; however, larger finger tables substantially decreased the operation times, as illustrated in Figure 3. For example, at 32 nodes, the average time per operation for $M = 8$ was approximately 0.12 seconds. For $M = 0$, meaning no finger table was used, the average time per operation was close to 0.45 seconds.

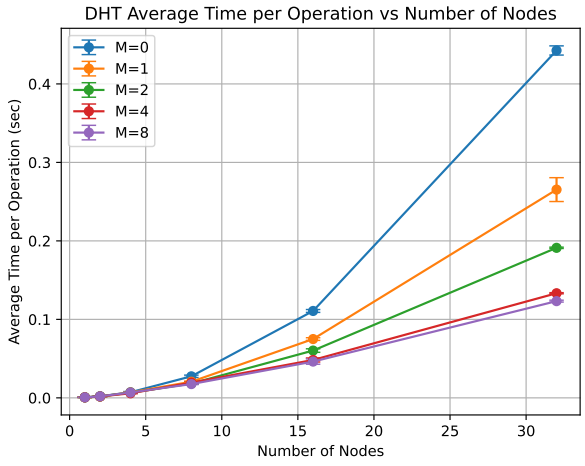

Fig. 3. Average time per operation measured with an increase in the number of nodes and finger table sizes.

VII. Discussion

The experimental results demonstrate trends that align with the theoretical properties of the Chord protocol. As the number of nodes increased, the throughput of the DHT and the average operation time increased. This outcome is expected, since more nodes introduce additional routing steps and communication overhead. However, the results also confirm that larger finger tables improve efficiency, lower operating times, and increased throughput compared to smaller finger tables.

For example, at 32 nodes, throughput increased from only 72 ops/s with $M = 0$ to 260 ops/s with $M = 8$, while the average operation time decreased from approximately 0.45 seconds to 0.12 seconds. This difference illustrates the benefit of Chord's logarithmic complexity. With a sufficiently large finger table, lookups require at most $O(logN)$ hops, which substantially reduces latency compared to linear routing without a finger table.

Overall, these experiments confirm that our implementation of Chord behaves as expected: it scales logarithmically with the number of nodes, and larger finger tables reduce lookup cost by bringing performance closer to the theoretical $O(logN)$ complexity.

VIII. Conclusion

This report discussed the implementation and evaluation of a Chord-based distributed hash table. Our experiments demonstrated that throughput decreases as the number of nodes increases. However, larger finger tables significantly improve performance by reducing lookup times. These findings are consistent with the theoretical $O(logN)$ complexity of Chord, confirming that our implementation provides a scalable and efficient solution for a distributed hash table.

References

[1] I. Stoica, R. Morris, L.-N. David, D. M. 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.

[2] M. Van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed. Maarten van Steen, 2023.

[3] Wikipedia. Distributed hash table. Accessed on September 28th, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Distributed_hash_table

[4] Wikipedia. Chord (peer-to-peer). Accessed on September 28th, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Chord_(peer-to-peer)

[5] S. Dutta. Understanding distributed hash tables: An in-depth guide to chord. Accessed on September 28th, 2025. [Online]. Available: https://medium.com/@sutanu3011/uncover-the-mystery-of-chord-56f8d216b7e0

Appendix A
Use of Generative A.I.

Generative artificial intelligence tools were utilised to enhance the language and clarity of the text.

APPENDIX B
TEST RESULTS

TABLE I
THROUGHPUT BY NUMBER OF NODES AND FINGER COUNT M

Nodes	M	Throughput (ops/s)	Std. Dev.
1	0	1533.57	18.91
1	1	1554.06	32.96
1	2	1529.68	19.43
1	4	1554.29	36.22
1	8	1565.81	30.21
2	0	1031.15	93.00
2	1	1194.62	154.07
2	2	1077.99	45.65
2	4	1111.27	61.42
2	8	1047.89	12.34
4	0	559.40	70.95
4	1	572.07	38.71
4	2	615.78	25.19
4	4	690.06	68.03
4	8	586.96	30.86
8	0	291.48	16.10
8	1	389.33	16.05
8	2	438.38	22.58
8	4	412.70	17.31
8	8	455.43	16.59
16	0	144.51	2.37
16	1	213.55	4.79
16	2	265.51	9.81
16	4	331.52	17.52
16	8	346.19	25.12
32	0	72.30	0.96
32	1	120.60	6.91
32	2	167.39	0.73
32	4	240.06	1.30
32	8	259.65	2.52