Distributed Hash Table and the Chord protocol

Assignment 2 INF-3200 Distributed systems

Iver Mortensen
imo059@uit.no

Theodor Tredal
ttr042@uit.no

I. Introduction

Chord is a peer-to-peer protocol that implements a Hash table (DHT) to efficiently locate and store data across a decentralized network. As the number of nodes or the key space grows, locating a specific key using naive methods can become inefficient or infeasible. Chord addresses this by organizing the nodes into a logical ring and using a distributed routing mechanism to quickly determine which node is responsible for a given key. This report presents a simulation of the Chord operations, including node lookups and key insertions, where nodes work together to collaborate and retrieve keys. The report describes how each version is implemented, how node routing and finger tables are implemented, and presents performance experiments measuring lookup times and scalability as the network size increases. Finally, the reports discuss strengths and weaknesses of the approaches.

II. Technical Background

A. Distributed hash table (DHT)

The distributed hash table, known as DHT, is a distributed data structure used to store and retrieve keys and values in a network of computers without the need for a centralized server. A hash table is a data structure, where a value is connected to key, normally this table of key-value pairs is stored on a single table, in a DHT this table is distributed among several nodes in a shared network, each node is responsible for their own part of the key space.

B. The CHORD Protocol

CHORD is a peer-to-peer protocol that implements a DHT to efficiently retrieve and store data in a decentralized network. The main problem that CHORD aims to solve is resource discovery; given a key how a system quickly determine which node is responsible for storing the corresponding value without the need for a centralized server?

The core idea behind CHORD is to arrange all nodes in the network into a logical ring, known as the identifier circle. Each node is assigned a unique identifier by hashing its IP address, using a hash function such as SHA-1, the same hash function is also used to generate identifiers for keys. This creates a keyspace of size 2^m where m is the number of bits in the hash function (e.g., $m = 160$ for SHA-1).

Each key belongs to the node with the smallest identifier that is equal to or larger than the key's identifier. This node is known as the key's successor. In addition, each node maintains information about its immediate predecessor and successor, ensuring that the ring structure is preserved even as nodes join or leave the network.

To efficiently support lookups, Chord uses a routing mechanism based on a finger table. Instead of forwarding requests step by step to each node in the ring, each node maintains pointers to nodes at exponentially increasing distances. This allows lookups to be performed in $O(\log N)$ Time, where N is the number of nodes in the distributed network.

Although dynamic joining and leaving is not implemented in this assignment, Chord uses a stabilization protocol that handles the dynamic nature of peer-to-peer networks. The stabilization protocol ensures that the ring and finger tables remain consistent as nodes join, leave or fail.

C. Remote Proceduce Call

Remote Procedure Call (RPC), a mechanism that allows a program to call a function or a procedure on a remote computer as if it were a local function. RPC abstracts away the underlying network communication, handling the details of sending requests and receiving responses between machines.

III. Design & Implementation

A. Design

The chord ring layout can be seen in figure [1]. Each node is aware of the closest succeeding node, referred to as its successor. A node is responsible for the keyspace between itself and its predecessor, not including its predecessor. As an example, node 33 from figure [1] is responsible for the keyspace (18, 33]

The keyspace size, or the number of unique keys in the chord ring, is determined by the variable m. The keyspace size is calculated with the following formula.

$$\text{keyspace size} = 2^m$$

The chord ring in figure [1] has m=6, resulting in a keyspace of 64 different keys, starting at 0 to 63.

The SHA-1 hash function is used to calculate each nodes position in the keyspace. SHA-1 is also used to assign hash

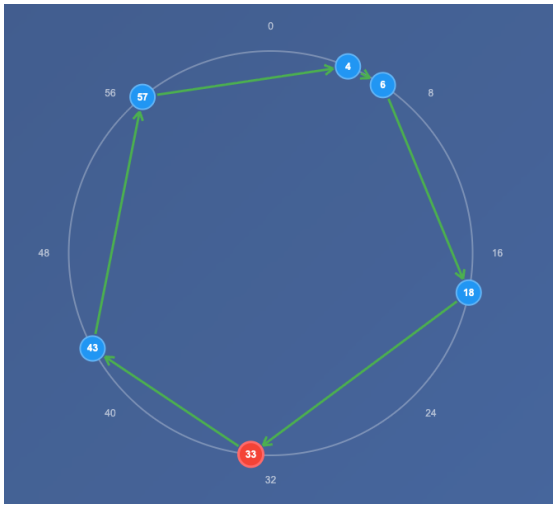

Fig. 1. Chord ring showing each chord node's successor.

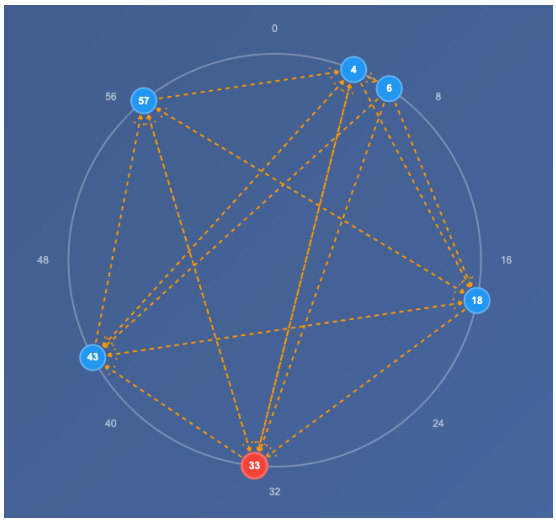

Fig. 2. Chord ring showing each chord node's finger table entries.

table keys to the same keyspace, which results in a consistent hashing, allowing any node to locate a key value pair within the chord ring.

B. Insertion and retrieval

When a client wants to use the distributed hash table of the chord ring, they start by connecting to a arbitrary node within the ring. If they want to store a value, they then send the key value pair to the node. The node forwards the key value pair to the responsible node, which then stores the pair in its portion of the DHT. When accessing a value, the same process is applied, the node forwards the key to its responsible node, which retrieves the value, returns it to the client's connected node, which then delivers the value to the client.

C. Finding the responsible node

When a node receives a request with a key, it starts by checking if the key resides within the keyspace of its successor. If it does, the owner of the key is found and the request can be handled. If the key doesn't reside within the node's successor, the search for the key's owner is forwarded to the successor node. The successor does the same procedure by checking its successor. This is done until the node responsible for the key is found. This setup is comparable to a linked list, and should produce the same lookup time complexity of O(n).

D. How to the finger table is implemented.

To improve the look-up time of the chord ring, a finger table for each node is used. The finger table contains nodes at different intervals along the chord ring. The formula to calculate each entry within the finger table is:

$$\text{finger}[i] = \text{successor}(n + 2^{i-1})$$

The Chord ring which implements finger tables can be seen in figure [2]. The finger table entries works as shortcuts within the chord ring. When a node wants to find the owner of a key, it first checks the last entry of the finger table, which will be the node that is furthest away. The node will work its way up the finger table until it finds a known node that is between itself and the key. The last node it checks will be its successor. The way that the finger table is constructed, makes the search similar to a binary search, as the search space is about halved at each step. This should allow for a time complexity of O(log n).

E. Implementation

The Chord ring is initialized using a bash script. The bash script starts by finding available nodes across the cluster. If not enough nodes are available it selects duplicates. The script also selects a random port for each node. The script then calculates the ID for each node, which will be the nodes location in the keyspace of the chord ring. Using the IDs of the nodes, successors of the nodes are also calculated. By using scp and ssh, the script sends each node a copy of the Chord code, and starts the Chord program on all the nodes.

When all nodes have been initialized, the script sends each of the nodes its successor. When all nodes have a successor, the chord ring is deemed functional, and works like the chord ring in figure [1]. The script ends by sending a request to all the nodes to update their finger table. Since all nodes have a successor and the chord ring is functional, the nodes can manage their finger table without external help.

The Chord server code implementation is done in Python, the communication between nodes uses the RPC library called Pyro5. Each node has a class which contains all attributes relevant to the node. This includes its ID, successor, finger table and local hash table (a Python dictionary). It also implements methods for finding closest known successor to a key, insert and retrieve values in its local hash table, and update its finger table. Each successor and finger table entry is a RPC reference to the specific nodes chord node class.

Each Chord node also runs an HTTP server. This server is separate from the RPC server which is meant for interal communication between the nodes. The HTTP server is used

for client and management interaction. It implements GET and PUT operations for managing the chord rings distributed hash table. It also contains the operations used by the run script to update a nodes successor, or to tell a node to update its finger table. Status operations for checking a nodes successor or finger table are also implemented.

The function below is the consistent hash function used to map a value to the keyspace.

$$\text{key} = \text{SHA-1}(\text{value}) \bmod 2^m$$

In the function the value is what we want to map to the keyspace, and the key is the location within the keyspace. When mapping a node to the keyspace, a hostname:port combination is used as the value. The hostname can be replaced with IP-addresses, and the port is optional. When mapping a hash table entry to the keyspace, the hash table key is used as the value.

IV. Experiments

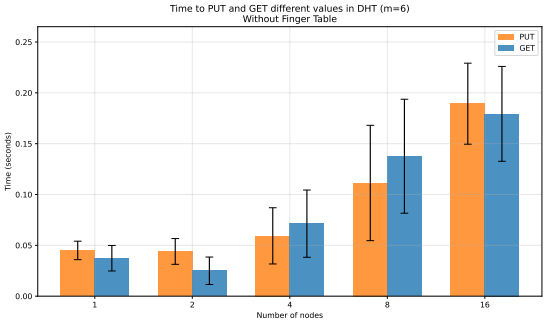

Fig. 3. Time to PUT and GET different values from the chord ring's DHT, without the finger table.

The figure [3] show the results of testing the DHT functionality of the chord ring, without the use of the finger table. The test was done in two parts. First by connecting to 3 random nodes in the chord ring and inserting (PUT) a different value into the DHT. The second step was to connect to 3 new random nodes, and access (GET) the 3 values in the DHT. This two step process was done with 1, 2, 4, 8 and 16 nodes. The figure shows the standard deviation of the 3 GET and PUT operations, for the different number of nodes.

Figure [4] shows the result of the second test on the DHT functionality of the chord ring. The test was executed the same way as the first however, the finger table was now used.

The two tests, although their simplicity, seems to align with the time complexities proposed earlier. The figure [3] seems to show a linear increase with a constant of around 0.05-0.025 seconds. Figure [4] have a better than O(n) time complexity in this limited test, which could indicate a O(log n) time complexity. More testing is required to derive anything concrete.

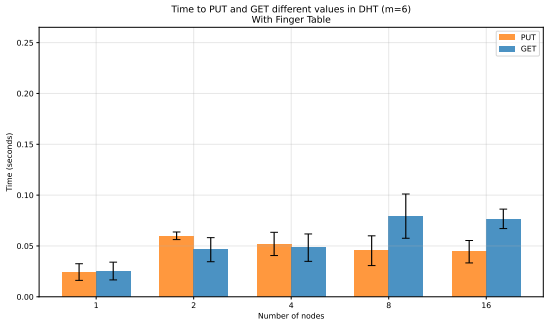

Fig. 4. Time to PUT and GET different values from the chord ring's DHT, with the finger table.

V. Discussion

On the computing cluster used during the assignment, each node has its own unique host name, and a limited amount of computing nodes. This meant that a computing node could end up running multiple chord nodes. Adding the port to the hash function meant that chord nodes running on the same computing node could have differentiating keys within the keyspace. This would not be the case if only IP-address was used, as the IP-address would be the same for all chord nodes running on the same computing node, resulting in identical keys after the hashing function. An upside to adding the port to the hash function input, is flexibility in node ID collision handling. When only using IP-addresses or host names, collisions can occur. The set of IP-addresses is greater than the set of keys in the key space. Adding the port as well grants some flexibility, as ports can be changed.

VI. Conclusion

This report presented an simple implementation and evalutaion of the Chord protocol and its distributed hash table. Nodes were organized in a logical ring, with keys assigned to responsible nodes and finger tables used to improve lookup efficience. RPC enabled inter-node communication, while an HTTP interface allowed client interactions. Experiments showed that without finger tables, lookup times grow linearly, whereas using finger tables significantly improved performance, approaching logrithmic behaviour. The assignmen demonstrates the main prinsiple of decentralized data storage, efficient routing and key responsibility. Though outside the scope of this assignment, further testing on a larger network could provide additional insights into scalability and the robustness of the Chord protocol.

References

VII. Sources

References

[1] Stoica, Ion, et al. (2001, August). Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. In *ACM SIGCOMM Computer Communication Review*, 31(4), 149–160. Retrieved September 22, 2025, from https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1180543&tag=1

[2] Wikipedia contributors. (2025 August 9), Chord (peer-to-peer). In Wikipedia, The Free Encyclopedia. Retrieved September 24, 2025, from https://en.wikipedia.org/wiki/Chord_(peer-to-peer)

[3] Wikioedia contributors. (2025 September 20), Remote procedure call. In Wikiepdia, The Free Encyclopedia. Retrieved September 23, 2025 from https://en.wikipedia.org/wiki/Remote_procedure_call