INF-3200 Distributed Systems

Assignment 2 - Distributed Hash Table

Magnus Dahl-Hansen

September 2025

1 Introduction

The goal for this assignment was to develop a distributed hash table (DHT) which is a system for storing key-value pairs among multiple nodes in a network. Specifically, the DHT was implemented using the Chord protocol which maps nodes and keys onto an identifier circle by using consistent hashing. This report describes how the Chord protocol was implemented and how it was tested in detail.

2 Implementation

This implementation can be separated into four main parts:

- Launching nodes and starting the program
- Initialization of the chord ring
- Implementing nodes that can respond to client requests to store and retrieve key-value pairs
- Implementing a client process that sends requests to inserts and retrieve values from the distributed hash table

Upon running the program, the user can specify how many nodes the Chord ring will contain. A dedicated shell script retrieves the names of available nodes in a cluster, creates a port number for each node and launches them. The script then runs a Python program that initializes the Chord ring and provides each node with necessary configuration information.

To initialize the ring, each node is assigned a unique ID which identifies them, a finger table that contains the IDs and IP addresses of a subset of other nodes in the network and pointers to its closest predecessor node and successor node. The unique ID for each node is created by running their IP address through a SHA-1 hashing function. Each ID is added to a list, which is further

used to create a unique finger table for each node. When all the IDs and finger tables have been created, they are each sent to their respective node via a HTTP POST request. Once all nodes are initialized, the shell script launches a Python client program and provides the client with a single node IP address that it can contact.

The client program uses a for-loop to create 100 different integer key-value pairs and sends them to the only known node via HTTP PUT requests. After a short delay it will send 100 HTTP GET requests with the same generated keys to try and retrieve the corresponding values. The client also implements two timers that measures the time used to send 100 values and retrieve 100 values. These timers will later be used to calculate the throughput of the system for different network sizes.

Each node is implemented with four HTTP routes that triggers different functions:

- Initialize()
- Network()
- PUT-data()
- GET-data()

A PUT request to *initialize* provides the node with its unique ID, finger table, successor ID and predecessor ID, which it proceeds to store for later use.

A GET-request to *Network* returns a list of the IDs which the current node has in its finger table.

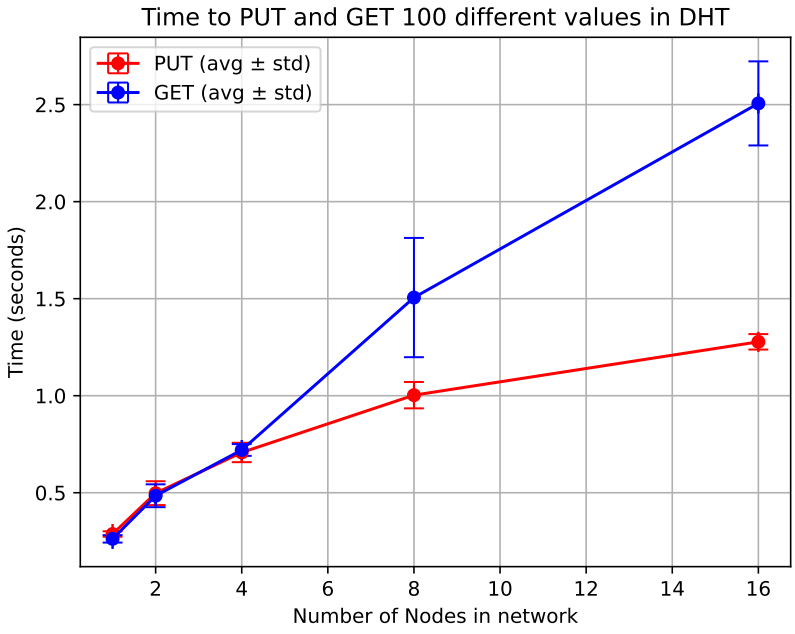
When a PUT-request to *PUT-data* is made, a key-value pair is sent to the node. The key is hashed using SHA-1 and mapped into the circle using the modulo operator, this way we can find which node ID it should be stored in. The node then performs three checks: wether the current node is responsible for storing the key-value pair, wether they should be forwarded to its successor node or if they should be forwarded to the closest node in the finger table. If the current node stores the key, it will respond with HTTP "OK" 200. If the request is forwarded then the node will wait for a response and return the same response it receives, effectively allowing the main response to be chained back to the client.

Similarly, when a GET request is sent to *GET-data* only a key is provided. The key is hashed using SHA-1 and mapped using the modulo operator to figure out which node ID in the circle is responsible for having the data. Three checks are again made to see if the current node should be the holder of the requested data, or if the request have to be forwarded to its successor or its closest node in the finger table. When the request finally arrives at its correct node, a HTTP

"OK", 200 response is sent if the data exists, and a HTTP "Not found", 404 will be sent if the data does not exist. The response is then relayed through the chain of nodes until it finally reaches the client.

3 Experiment

The implementation have been tested with different network sizes (i.e., the number of nodes in the Chord ring) and the throughput of the system have been measured in how many PUT and GET operations it can handle per second. The results as illustrated in the plot below, shows that as the number of nodes increases, throughput decreases for both PUT and GET operations. This happens because a larger network creates more ID spaces where data can be stored or retrieved from, and therefore also creates more forwarding requests. When the ring consists of a few nodes, each node is responsible for larger ranges of keys IDs, so requests are more likely to be handled by few nodes. When the ring consists of more nodes, logically each node is responsible for smaller key ranges and therefore more forwarding requests has to be made. Forwarding increases the time used in two ways: 1) each time a message has to be forwarded it adds latency as HTTP requests takes time. 2) every time a request is sent, the key is hashed, the finger table has to be searched to find the correct node and the response has to be sent through the same chain of nodes. This explains why we can see that the system performs best when only 1-4 nodes are used compared to when 8-16 nodes are used.



It is also worth noting that the graph shows that GET operations takes nearly twice as much time as the PUT operations. A few reasons for this could be:

1. A PUT operation only needs to hash a key, send it to the correct node and store it. A GET operation has to do the same while also checking if the data actually exists in storage.

2. A PUT request only has to carry data one way, then reply with a small response such as "OK". In a GET request a key has to be sent and the actual value has to be in the response. Transferring that payload over many nodes adds latency.

4 Discussion

During testing, an error appeared a few times when running the system with 8 or more nodes. In these cases the system appeared to go into an endless loop. This is likely due to an implementation error in how the ID ranges are assigned for each node. As the ID space grows, there could be some gaps where no nodes are responsible for certain IDs. In this case each node keeps forwarding the message to the closest node they think is responsible for the ID and the request never arrives to the correct node, ultimately creating an infinite loop. This is

issue becomes more notable as the number of nodes in the network increases.

Given more time, I would have liked to try and optimize this implementation in the following ways:

1. In the current implementation each node re-hashes the key every time it is recieved, wheter it is a PUT or GET request. A more efficient approach would be to hash the key only once either at the client or at the first node, then send the hashed key through the network. This way, less time would be spent hashing the key at every step.

2. Currently, responses from the nodes are sent back in the same way they arrived. To get rid of this channeling the clients IP address could be included in the message so when the request was handled at the correct node, it could send its response message directly to the client. This would contribute to reduce network latency.

3. Implementing a check to see wether a key already exists in a nodes memory during a PUT request. This would not optimize the performance, but would prevent accidental overwrites.

5 Summary

In this assignment a distributed hash table using the Chord protocol has been implemented and tested. The system consists of multiple nodes organized in a Chord ring, where each node is responsible for storing and retrieving the values mapped to specific key IDs. The system supports GET and PUT requests from a client, which are forwarded between the nodes in the system until they reach the correct destination.

The system has been tested by measuring the time it takes to handle 100 PUT and 100 GET operations with different network sizes ranging from 1-16 nodes. The results showed that the throughput decreased as the number of nodes in the network increased. This is primarily because there is more forwarding of requests, increased network latency, more hashing operations and searches through the finger tables. Additionally, there has been observed that GET operations are significantly slower than PUT operations, most likely because of that retrieving a value results in more look ups in the hash tables and bigger data transfers.

This implementation can successfully store and retrieve values in a distributed hash table system. However, in some cases when the network consists of more than 4 nodes, infinite loops may occur because of issues with how ID ranges are assigned and managed at each node.