INF-3200: Distributed System Fundamentals Assignment 1: DHT and the Chord protocol

Ilya Taksis & Asbjørn
UiT id: ita022@uit.no
UiT id: aaa040@uit.no October 5, 2023

1 Introduction

This assignment focuses on creating a distributed key-value store in the form of a distributed hash table (DHT) using the Chord protocol. This report will cover the design and implementation of this DHT and the testing, evaluation, and measurement of the system's throughput.

2 Technical background

2.1 Chord protocol

The Chord protocol is a distributed lookup protocol that supports one operation: given one key, it maps the key onto a node[1]. The Chord protocol achieves its functionality through several key mechanics. Consistent hashing is used to assign identifiers. The key identifiers are assigned an m-bit identifier by using the SHA-1 hash function. The node identifiers are assigned by hashing their IP addresses. A key location, is where the node that is the successor of a key is the node responsible for that key. Finger tables, where each node maintains a table with up to m entries, and each entry in the table contains the ID of the first node that succeeds the key $n + 2^{i-1}$ on the identifier circle. This table enables Chord to have on average a logarithmic lookup time even as the network scales.

3 Design and Implementation

This section covers the design and implementation of the DHT and the Chord protocol. This section is also split up into 2 subsections, where the first covers the initialization of the node and the other subsection covers the different APIs.

3.1 Node initialization

Before starting the node server, the node must undergo initialization, including setting its correct IP address, node ID, responsible keys, and finger table. This process involves:

Obtaining the node's local IP address by creating a socket and reading its IP from it. The node's ID is generated by hashing its local IP using a consistent hashing function, specifically SHA-1, and then performing a modulo operation with the total cluster size to ensure the Distributed Hash Table (DHT) forms a ring structure with keys and nodes.

Once the node has its IP address and node ID, it proceeds to construct its finger table. This entails taking the node ID and the total number of nodes in the Chord ring and parsing relevant node information from an "ip.txt" file, which contains numerous nodes. The parsed nodes are sorted by their node IDs before filling in the finger table. The finger table entries are computed using the formula $n + 2^{i-1}$, where n is the current node

ID, and i is the index of the finger table. This calculation determines succeeding keys, and the corresponding responsible nodes are identified by iterating through the list of parsed nodes.

Next, the node determines its previous neighbor to determine the keys for which it is responsible. To find the previous neighbor, it repeats the process of parsing nodes from "ip.txt," sorting them and initializing the previous node as the last node in the cluster. It then iterates through the sorted nodes and updates the previous node variable whenever it encounters a node with an ID less than the given node ID. This approach effectively identifies the previous node in the sorted cluster based on ID.

Lastly, the node can find what keys it is responsible for by iterating over the range from the previous node ID to its current ID, counting all the key IDs that are between them, and saving them in its list. But if the previous node ID is bigger than the current node ID, it first counts from the previous node ID to the cluster size, and then it counts from 0 to the current node ID because the DHT is a ring. Now that everything is set up and initialized, the HTTP server can start.

3.2 APIs

Every node handles three distinct API calls:

GET API Call ("/storage/neighbors"): This call is designed to return the neighboring nodes of the requesting node. It responds with a JSON object containing the previous node's IP address and the IP address of the first node in the finger table, along with an HTTP 200 status code.

GET API Call ("/storage/key"): This call is intended to retrieve the value stored under a given key. It involves hashing the provided key with a consistent hashing function and checking if the current node is responsible for the specified key ID. If it is responsible, the node accesses its hashmap to find the corresponding value. If the key is present in the hashmap, it responds with an HTTP 200 status code and the associated data. However, if the key is not in the hashmap despite the node being responsible for the key ID, it returns an HTTP 404 "NotFound" status code. If the current node is not responsible for the key ID, it locates the first successor and determines if they are responsible. If they are, the node forwards the key to the first successor via an HTTP PUT request. If the first successor is also not responsible, the node iterates through its finger table to identify a responsible node. If found, it forwards the key to that node. If no suitable node is identified in the finger table, the node forwards the key to the node with the highest node ID in the finger table.

PUT API Call ("/storage/key"): This call involves storing data associated with a specific key. It follows a similar process as the second API GET call regarding hashing the key and checking responsibility. If the node is responsible for the key, it stores the key and the provided data in its hashmap and responds with an HTTP 200 status code. If the node is not responsible, it identifies the successor node using a process similar to that of the second API GET call, and the responsible node responds with an HTTP 200 status code after successfully storing the data.

In summary, nodes handle API calls for neighbor information, key retrieval, and key storage, forwarding requests as needed to ensure proper operation within the Chord ring.

4 Experiment and Analysis

In this section, we present the results of our experiments conducted to evaluate the performance of the Chord distributed hash table system under varying network node configurations. We conducted 10 runs for each of the following node configurations: 1, 2, 4, 8, and 16 nodes. Our primary metric for evaluation was throughput, measured in requests per second (RPS).

4.1 Throughput Analysis

As shown in Figure 1, the experiment results show a clear trend in the system's throughput as the number of nodes in the Chord network varies. Starting with a single node in the network, the system exhibits a high throughput of approximately 600 RPS. However, as we introduce additional nodes, the throughput decreases logarithmically. With two nodes, the throughput drops to around 80 RPS, and this trend continues as we scale up the network. At 16 nodes, the throughput reaches its lowest point, with a throughput of approximately 15 RPS.

4.2 Request Type Analysis

We also examined the behavior of two different types of requests: "get" and "put." Figure 1 illustrates the variations in throughput for these request types across the different node configurations. Our experiment reveals that while there are some differences in throughput between "get" and "put" requests, these differences are almost negligible in the context of the overall throughput degradation as the number of nodes increases. This implies that our implementation performance is impacted more by the increased network complexity associated with a larger number of nodes rather than the specific request types.

4.3 Discussion

The observed logarithmic decrease in throughput as the number of nodes in the Chord network increases shows the challenges of maintaining high performance in a distributed hash table system. This decrease is consistent with the increased routing complexity and communication overhead inherent in larger networks.
The negligible differences in throughput between "GET" and "PUT" requests show the system's ability to handle both read and write operations efficiently, regardless of the node configuration. This suggests that the performance bottleneck primarily arises from the network structure and routing protocols of the Chord system itself, rather than the specific operations it performs.
In conclusion, our experiments provide some insight into the performance characteristics of the Chord distributed hash table system under different node configurations.

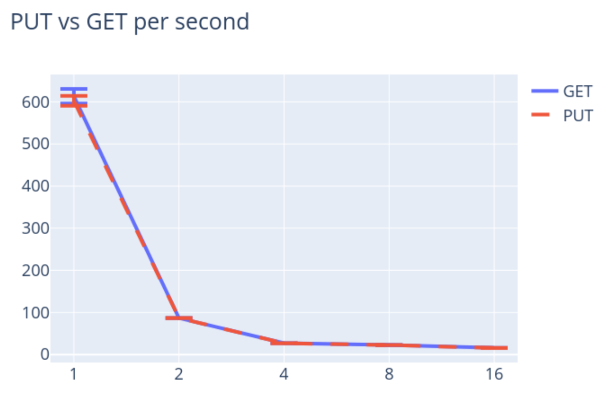

Figure 1: Requests per second 1 to 16 nodes

5 Conclusion

In conclusion, this implementation has explored the Chord protocol, a distributed lookup protocol for decentralized systems. By leveraging consistent hashing, key location principles, and finger tables, Chord offers an efficient and scalable solution for distributed key-value storage. Its logarithmic lookup time and robustness make it a valuable tool in the realm of distributed systems, facilitating seamless and efficient data retrieval in large-scale networks.

References

[1] Stoica, I. et al. (2003) ‘Chord: A scalable peer-to-peer lookup protocol for internet applications’, IEEE/ACM Transactions on Networking, 11(1).