INF-3200: Assignment 1b – Chord

Aditya Ponnappan

September, 2024

1 Introduction

In this assignment, we were tasked with implementing a distributed key-value store using a Distributed Hash Table (DHT) based on the Chord protocol [1]. Our system enables efficient storage and retrieval of key-value pairs across multiple interconnected nodes. Additionally, an experiment was conducted to measure the system's throughput, examining its scalability. This report is based on the successful submission of the same assignment by the author last year.

2 Technical Background

2.1 Distributed Hash Table (DHT)

A DHT is a decentralized system designed for distributed lookup and storage, much like a traditional hash table but operating across numerous network nodes. The main challenge in DHTs is efficiently distributing key-value pairs across nodes and establishing a protocol for routing requests to the appropriate node responsible for a given key.

2.2 Consistent Hashing

In a DHT, consistent hashing ensures balanced load distribution by efficiently spreading key-value pairs across the network, minimizing the risk of overloading specific nodes. It also allows the system to accommodate changes (such as node additions or removals) with minimal re-mapping of data.

2.3 Chord Protocol

Chord is a DHT protocol that maps nodes and keys onto a circular identifier space, referred to as the "hash ring", using cryptographic hashing [1]. In Chord, both keys and nodes are assigned positions on this ring based on their hash values. Chord uses a data structure known as the *finger table* to optimize the routing process. Each node only needs to be aware of a small subset of other nodes, and can efficiently route queries by traversing the ring in $O(\log N)$ hops, where N is the number of nodes in the network [1]. This design allows Chord to scale effectively while providing reliable data lookup.

3 Design & Implementation

3.1 Finger Table Initialization

The `initializeFingerTable` method plays a key role in setting up the necessary data structures for routing keys within the network. It performs the following tasks:

- **Sorted Hash Ring Setup**: It calculates the SHA-1 hash for each node's address and sorts the hash values to form a ring structure, representing the DHT's node distribution.
- **Neighbor Identification**: The function identifies a node's immediate neighbors (predecessors) on the ring, which is critical for routing.
- **Finger Table Setup**: Each node's finger table contains 10 entries, corresponding to 2^i-th successors on the ring, where i ranges from 1 to 10. This setup ensures efficient routing by allowing each node to maintain information about its neighboring nodes.

3.2 Successor Node Identification

To store or retrieve key-value pairs, Chord identifies the node responsible for a key, known as the successor node. The system first checks if the current node holds the key. If not, the query is forwarded to the successor node, or if necessary, to a node identified via the finger table, ensuring efficient routing. The process is explained in depth in the following sub sections.

3.3 Identifying the Forwarding Address in the Finger Table

In a Chord Distributed Hash Table (DHT), nodes maintain a finger table to facilitate efficient routing of requests for key-value pairs. The finger table provides information about other nodes in the network, which is crucial for forwarding requests when a node is not the responsible one for a given key.

3.4 Structure of the Finger Table

Each node's finger table consists of several entries (10), where each entry i points to a node that is a certain distance away in the logical ring. Specifically, the i^{th} entry of a node n in the finger table points to the first node that appears in the ring whose identifier is greater than or equal to:

$$n + 2^{(i-1)} \tag{1}$$

This means:

- Entry 1: Points to the node whose identifier is the smallest among all nodes that are greater than or equal to $n + 1$.
- Entry 2: Points to the node whose identifier is the smallest among all nodes that are greater than or equal to $n + 2$.
- And so forth, up to Entry 10.

This structure allows a node to have a logarithmic number of pointers to other nodes, making it possible to skip over many nodes in a single routing step.

3.5 Request Handling and Forwarding Logic

When a node receives a request (either PUT or GET) for a key, it needs to determine if it is responsible for that key based on the hashed value of the key. If it finds that it is not the responsible node, it must identify the correct node to forward the request to. Here's how the process works:

1. **Hashing the Key:** The key is hashed using SHA-1, resulting in a key identifier. This identifier is what the node will use to determine the responsible node.
2. **Range Check:** The node first checks if the hashed key falls within its own range. This is done by comparing the key identifier with its own identifier and that of its predecessor in the ring. If the hashed key is greater than the identifier of the predecessor and less than or equal to its own identifier, then this node is responsible for the key, and it will store the value. If not, it proceeds to the next step.
3. **Utilizing the Finger Table:**
   - The node then uses its finger table to find the closest node that is likely to be responsible for the key.
   - It iterates through its finger table entries in reverse order (from the highest entry to the lowest) to find the first entry whose identifier is greater than or equal to the hashed key. This strategy ensures that the node forwards the request to the nearest node that is greater than the key, reducing the number of hops needed.
   - For each finger table entry i:
     - If $finger[i]$ (the identifier of the node pointed to by entry i) is greater than or equal to the hashed key, then this node becomes a candidate for forwarding.
     - The search continues until the node identifies the largest i such that $finger[i]$ is less than or equal to the hashed key. This helps to ensure that the request is sent to the most appropriate node.

4. **Forwarding the Request:**
   - Once the node identifies the appropriate node from its finger table, it forwards the request to that node.
   - If the identified node is not responsible for the key, it will repeat the same process until the request reaches the correct node responsible for storing the key.

3.6 Request Handlers

- **PUT Request Handler**: This function handles HTTP `PUT` requests by determining the appropriate node to store a given key-value pair based on the key's hash. If the current node is responsible, the key-value pair is stored locally. Otherwise, the request is forwarded to the correct node via the finger table.
- **GET Request Handler**: The `GET` handler retrieves values from the DHT, following a similar process. It checks if the current node holds the requested key; if not, it forwards the request to the responsible node.

This design theoretically supports decentralized and fault-tolerant key-value storage, enabling efficient `GET` and `PUT` operations.

4 Experiments and Results

Experiments were conducted to measure the system's throughput, defined as the number of requests (`PUT` or `GET`) the system can process per second. They were measured across networks with 2, 4, 6, 16, and 32 nodes, running 10 trials for each network size. The results are shown in Figure 1.

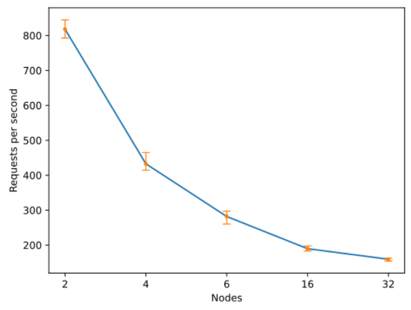

Figure 1: Throughput of Chord Implementation

4.1 Observations

Throughput decreases as the number of nodes increases, as expected. With more nodes, requests take longer due to potential hops between nodes. However, since the number of hops is logarithmic ($O(\log n)$), the throughput decline flattens out as the number of nodes increases. A possible improvement would be to enable clients to send requests in parallel, which could increase throughput.

5 Discussion

One limitation of our design is the possibility of hash collisions, where two nodes could share the same hash key. To minimize this risk, a hash ring size of $2^{10} = 1024$ was used, which should provide sufficient space to avoid collisions in most cases.

6 Conclusion

In this assignment, a scalable, decentralized key-value store using the Chord protocol was successfully implemented. Experiments were conducted to measure system throughput, finding that the system performs as expected, with throughput decreasing logarithmically as nodes are added.

References

[1] I. Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications." *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, 2003, pp. 17-32. doi: 10.1109/TNET.2002.808407.