Distributed Hashing - Final Report

Aayush Kumar (CO21BTECH11002)
Vishal Vijay Devadiga (CS21BTECH11061)

Table of Contents

Introduction

What is Distributed Hashing?

Distributed hashing is a technique used in distributed computing environments to efficiently store and retrieve data across multiple nodes in a network. It combines the principles of hashing, which involves mapping data to a fixed-size value, with distribution, where data is spread across multiple nodes or machines.

Here's how it typically works:

1. **Hashing**: Data is hashed using a hash function, which generates a unique identifier (hash) for each piece of data. This hash serves as the key for storing and retrieving the data.
2. **Distribution**: The hashed data is distributed across multiple nodes in the network. Each node is responsible for storing a subset of the hashed data based on a predetermined mapping scheme.

Distributed hashing is used in various distributed systems and applications for several reasons:

1. **Scalability**: It allows systems to scale horizontally by adding more nodes to the network. As the amount of data increases or the workload grows, additional nodes can be added to handle the load, without affecting the overall performance.
2. **Fault Tolerance**: Distributed hashing provides fault tolerance by replicating data across multiple nodes. If a node fails or becomes unavailable, the data can still be accessed from other nodes, ensuring high availability and reliability.
3. **Load Balancing**: By distributing data across multiple nodes, distributed hashing helps in balancing the workload evenly across the network. This prevents any single node from becoming a bottleneck and ensures optimal performance.

Distributed hashing is used in various distributed systems and applications, including distributed databases, content delivery networks (CDNs), peer-to-peer (P2P) networks, and distributed file systems like Hadoop Distributed File System (HDFS) and Amazon S3.

A brief overview of how DHTs work

There are several distributed hash table (DHT) algorithms that implement distributed hashing, each with its own unique approach to storing and retrieving data in a decentralized network. However, most DHT algorithms share some common characteristics:

1. **Key-based Routing**: DHTs use a key-based routing mechanism to route messages between nodes in the network. Each node is assigned a unique identifier (ID) based on a hash of its IP address or some other identifier. When a node wants to find a specific piece of data, it uses the key's hash value to route the request to the node responsible for storing that data.

2. **Overlay Network**: Each node in the network is connected to a subset of other nodes, forming an overlay network that facilitates communication and data exchange. The overlay network is typically structured as a mesh, ring, or tree topology, depending on the DHT algorithm. Also each node is reponsible for a subset of the data based on the hash of the data.
3. **Distance Metrics**: DHTs use distance metrics to determine the proximity between nodes in the network. This proximity metric is different for each DHT algorithm and is used to optimize routing and data storage.
4. **Data Replication**: DHTs replicate data across multiple nodes to ensure fault tolerance and high availability. Each piece of data is stored on multiple nodes, typically using a replication factor to determine the number of copies.
5. **Data Retrieval**: When a node wants to retrieve a piece of data, it sends a request to the node that is visible to the node and is closest to the key. The request is forwarded through the overlay network until it reaches the node responsible for storing the data. The data is then retrieved and sent back to the requesting node.

Kademlia

Kademlia is a distributed hash table (DHT) algorithm that uses a binary tree-like structure to store and retrieve data in a decentralized network. It is designed to be scalable, fault-tolerant, and efficient, making it suitable for large-scale distributed systems.

Key features of Kademlia include:

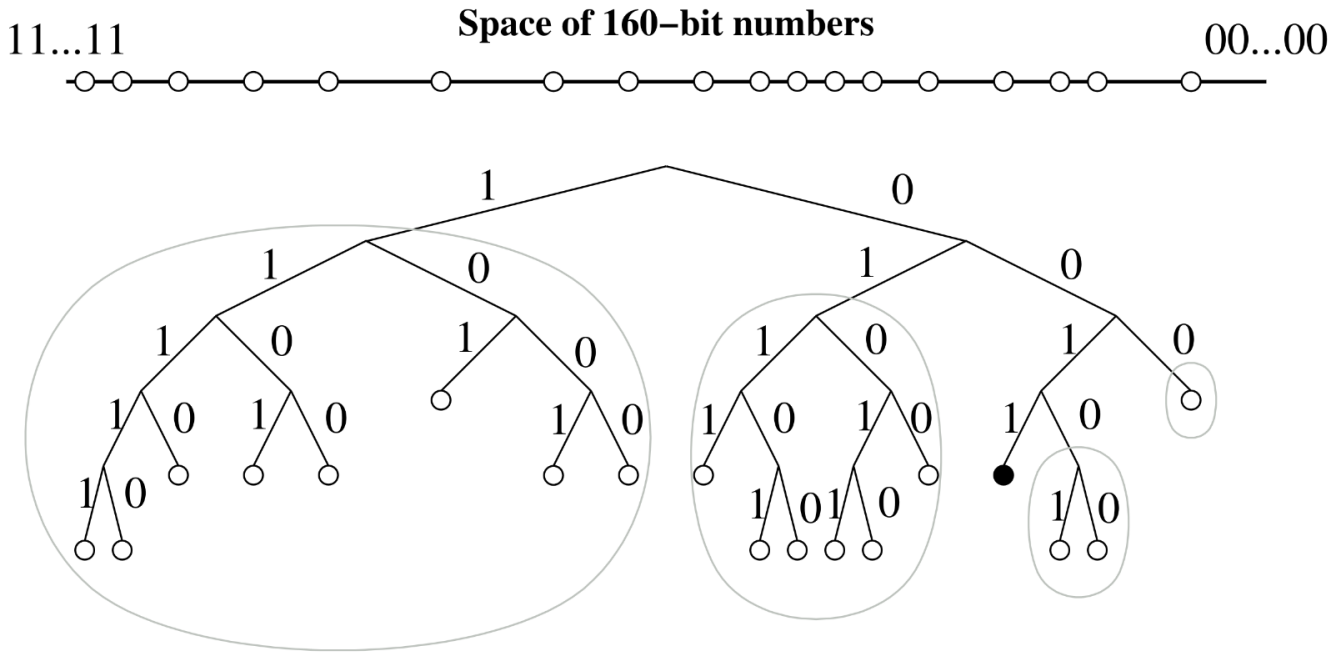

Figure 1: Kademlia Tree

1. **Binary Tree Structure**: Kademlia organizes nodes in a binary tree-like structure, where each

node is assigned a unique identifier (ID) based on a hash of its IP address. The binary tree is used to route messages between nodes and store data in a distributed manner.

2. **Distance Metric**: Kademlia uses the XOR metric to calculate the distance between two nodes in the network. The XOR metric is based on the bitwise XOR operation between two node IDs and is used to determine the proximity between nodes.

Suppose we have two nodes with IDs `A` and `B`. The distance between `A` and `B` is calculated as `A XOR B`, where `XOR` is the bitwise XOR operation. The node with the smallest XOR distance to the key is considered the closest node to the key. Let's say we have a key `K` and we want to find the node closest to `K`. We calculate the XOR distance between `K` and each node in the routing table and select the node with the smallest XOR distance as the closest node to `K`. Considering an example of a node with ID `01011101` and a key `10101010`, the XOR distance between the node and the key is `01011101 XOR 10101010 = 11110111`.

3. **Routing Table**: Each node maintains a routing table that contains information about other nodes in the network. The routing table is divided into buckets, with each bucket corresponding to a specific distance range from the node's ID. The routing table helps in efficiently routing messages and finding nodes that are close to a given key.

When the amount of bits in the address space is `n`, the routing table of a node is divided into `n` buckets, with each bucket corresponding to a specific prefix length. The bucket size determines the number of nodes that can be stored in each bucket. The routing table helps in efficiently routing messages and finding nodes that are close to a given key.

Example of the routing table of a node with ID `0101` when 4bit address space is used and the bucket size is 1:

Bucket Prefix	Nodes
1	1000
00	0001
011	0110
0100	0100

4. **Lookup Algorithm**: Kademlia uses a recursive lookup algorithm to find the closest node to a given key. The algorithm starts by querying nodes in the routing table that are closest to the key. If the closest node does not have the data, the algorithm recursively queries other nodes that are closer to the key until the data is found. The lookup algorithm is designed to be efficient and minimize the number of messages exchanged between nodes.

5. **Hashing Values**: The original paper uses 160-bit hashes to avoid collision and to ensure that the keys are uniformly distributed across the address space.

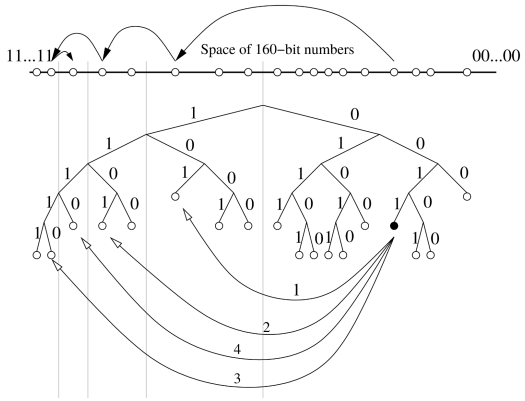

Figure 2: Kademlia - Lookup

Pastry

Report

Problem Statement

In this project, we implemented two distributed hashing algorithms - **Pastry** and **Kademlia** and evaluated the performance of these algorithms based on various metrics such as the number of hops, time taken, throughput, and load balancing and used them in a real-world application.

Approach

We first read the papers on Pastry and Kademlia to understand the working of these algorithms. On a high level, both algorithms use a similar approach to store and retrieve data in a distributed network. They use a key-based routing mechanism to route messages between nodes and store data in a decentralized manner. However, they differ in the underlying data structures, routing mechanisms, and distance metrics used to determine the proximity between nodes. We decided to extract common code in both the algorithms and then implement the specific parts of each algorithm separately.

Implementation

While coding the algorithms, we focused on the following key components:

- **Node ID Generation**: Each node is assigned a unique identifier (ID) based on a hash of its IP address or some other identifier. The ID is used to determine the position of the node in the network and to calculate the distance between nodes. We used `openssl` to generate random node IDs for the nodes in the network.
- **Routing Table**: Each node maintains a routing table that contains information about other nodes in the network. The routing table is used to efficiently route messages and find nodes that are close to a given key. We implemented the routing table as a data structure that stores information about other nodes in the network.
- **Message Routing**: When a node wants to find a specific piece of data, it uses the key's hash value to route the request to the node responsible for storing that data. We implemented the message routing mechanism to forward messages between nodes in the network.
- **Data Storage and Retrieval**: Each node is responsible for storing a subset of the hashed data based on the hash of the node. We implemented the data storage and retrieval mechanism to store and retrieve data in a distributed manner.
- **Lookup Algorithm**: We implemented a recursive lookup algorithm to find the closest node to a given key. The algorithm starts by querying nodes in the routing table that are closest to the key and recursively queries other nodes that are closer to the key until the data is found.
- **Evaluation Metrics**: We evaluated the performance of the algorithms based on various metrics such as the number of hops, time taken, throughput, and load balancing. We collected data on these metrics during the execution of the algorithms and analyzed the results to compare the

performance of Pastry and Kademlia.

Evaluation

We evaluated the performance of Pastry and Kademlia based on the following metrics:

1. **Number of Hops**: The number of hops required to route a message between nodes in the network. A lower number of hops indicates more efficient routing and faster data retrieval.
2. **Time Taken**: The time taken to route a message between nodes in the network. A lower time indicates faster data retrieval and better performance.
3. **Throughput**: The rate at which messages are routed between nodes in the network. Higher throughput indicates better performance and scalability.
4. **Load Balancing**: The distribution of data across multiple nodes in the network. Load balancing ensures that the workload is evenly distributed across nodes, preventing any single node from becoming a bottleneck.

These metrics help us evaluate the performance of Pastry and Kademlia in terms of efficiency, scalability, fault tolerance, and load balancing. We collected data on these metrics during the execution of the algorithms and analyzed the results to compare the performance of Pastry and Kademlia.

Graphs

1) Number of hops
2) Time
3) Throughput
4) Load balancing

Conclusion

In conclusion, we:

- Familiarized ourselves with the concept of distributed hashing
- Studied and implemented two distributed hashing algorithms - Pastry and Kademlia
- Evaluated the performance of these algorithms based on various metrics
- Analyzed the strengths and weaknesses of each algorithm
- Used both algorithms in a application to demonstrate their effectiveness in a real-world scenario

Reference

- Pastry Paper
- Kademlia Paper