Distributed Systems

Thodoris Paparrigopoulos

PhD Student

tpaparrigopoulos - 03003199

Table of Contents

Introduction

The Chordify project focuses on the design and implementation of a peer-to-peer (P2P) file-sharing application leveraging the Chord Distributed Hash Table (DHT) protocol. The goal is to create a scalable and efficient system that allows users to store and locate song metadata across a distributed network of nodes. By simulating real-world distributed systems, the project emphasizes core functionalities, including node management, data replication, and efficient routing without focusing too much on error handling. It also explores consistency models, such as linearizability and eventual consistency, to ensure reliable data storage and retrieval. This work provides practical insights into the operation and challenges of distributed systems, emphasizing the balance between performance and consistency.

Code Architecture

The codebase is structured into two primary components: the **Client** and the **Server**.

Client

Cli

The client provides a command-line interface (CLI) that allows users to interact with the local server by executing specific commands. These commands enable data operations, node management, and system inspection.

Available Commands

1. **Insert:**
   `insert <key> <value> [--host <host>] [--port <port>]`
   - Adds a new `<key, value>` pair to the system.
   - By default, `value` is set to `<host>:<port>` if not explicitly provided.
2. **Query:**
   `query <key> [--host <host>] [--port <port>]`
   - Retrieves the value for the given `<key>`.
   - If `key` is "*", it returns all `<key, value>` pairs from every node in the network.
3. **Delete:**
   `delete <key> [--host <host>] [--port <port>]`
   - Deletes the `<key, value>` pair from the system if the specified `value` matches.
   - By default, `value` is set to `<host>:<port>` to prevent accidental deletion from other nodes.
4. **Overlay:**
   `overlay [--host <host>] [--port <port>]`

    - Displays the current network topology, showing the logical arrangement of nodes in the Chord ring.
5. **Info:**
   `info [--host <host>] [--port <port>]`
    - Provides detailed information about the local node, such as its ID, neighbors, and stored data.
6. **Depart:**
   `depart [--host <host>] [--port <port>]`
    - Gracefully removes the node from the network, redistributing its data and updating the ring topology.
7. **Help:**
   `help`
    - Prints a summary of all available commands.

Command Processing

The client sends these commands as requests to the local server, which executes the corresponding operations. Each command maps to a specific server action:

- **Insert:** `PUT <key> <value>`
- **Query:** `GET <key>`
- **Delete:** `DELETE <key> <value>`
- **Overlay:** `GET_OVERLAY`
- **Info:** `GET_NODE_INFO`
- **Depart:** `DEPART`

This separation of the client and server ensures modularity and allows the server to run independently while the CLI facilitates user interaction.

Supporting Files

The client directory includes shell scripts for automating experiments and evaluating the system's throughput and performance:

1. **run_inserts.sh**
    - Automates the insertion of keys into running servers using input files.
    - Executes processes in parallel to simulate live system behavior.
2. **run_queries.sh**
    - Automates key queries across the servers using input files.
    - Executes processes in parallel to simulate live system behavior.
3. **run_requests.sh**
    - Performs general requests (insertions and queries) across the system.
    - Executes processes in parallel to simulate live system behavior.

4. **run_experiment.sh**
    - Combines the above scripts into a single workflow to execute all experiments seamlessly.

Server

The server is designed to manage the core logic of the Chord Distributed Hash Table (DHT) and handle communication with the client and other nodes in the network. The codebase is modular, with distinct files serving specific roles.

Main Components

1. **`main.py`**
    - Serves as the entry point for the application.
    - Configures the argument parser and initializes the main thread of execution.
    - Handles graceful shutdown of the server using `Ctrl+C` (SIGINT).
2. **`server.py`**
    - Manages the server's core functionality, including socket creation and thread management.
    - Initializes a socket server bound to a specified host and port.
    - Spawns a thread to accept incoming connections and processes requests.
    - Implements a custom protocol for transmitting and receiving large data chunks, using an 8-byte big-endian payload size header.
    - **Request Dispatching:** The `_dispatch` method maps incoming commands to corresponding functions in the `ChordNode` class. Commands include:
        - `GET_NODE_INFO`
        - `PUT`, `GET`, `DELETE`
        - `JOIN`, `DEPART`
        - `TRANSFER_KEYS`, `MOVE_ALL_KEYS`
        - `UPDATE_SUCCESSOR`, `UPDATE_PREDECESSOR`
2. **`chord_node_simple.py`**
    - Contains the core logic for the Chord DHT.
    - Manages node-specific operations such as data storage, key routing, and consistency handling.
    - Maintains:
        - `self.uploaded_songs`: Tracks songs uploaded by the node.
        - `self.data_store`: Stores key-value pairs in a specific format.

    - `self.successor`: Store a tuple of the next node.
    - `self.predecessor`: Store a tuple of the previous node.
- **Node Initialization:**
    - On startup, the `join` function is executed. The node contacts the bootstrap node, identifies its position in the ring, and exchanges keys with its successor using the `_acquire_keys` method.
- **Core Methods:**
    - `find_successor`: Identifies the primary node for a given key.
    - `chord_join:` A new node sends JOIN command to the bootstrap node, this command finds the successor and predecessor for that new node and send this information.
    - `chord_put`: Stores or updates a song in the DHT.
    - `chord_get`: Retrieves a song in the DHT.
    - `chord_delete`: Deletes a song if the key-value pair matches.
    - `depart`: Follows a process to remove the node from the DHT. Firstly, it deletes all songs that have been uploaded by this node. It notifies the successor through a UPDATE_SUCCESSOR about the predecessor of the deleting node. It also notifies the predecessor about its new successor through a UPDATE_PREDECESSOR command.
    - `chord_get_all`: Retrieves all songs in the DHT.
    - `chord_overlay`: Displays the current network topology, showing the logical arrangement of nodes in the Chord ring.
    - `chord_transfer_keys`: This is called when a TRANSFER_KEYS command is sent. This function is called from the _acquire_keys function, so essentially on the join of a new node. It is usually sent in the successor of a node to notify for the keys that belong to that node.
- **Help Methods:**
    - `chord_move_all_keys`: This function is being called on MOVE_ALL_KEYS commands. This command is used on a departure of a node. In this command the node sends out to its successor all of its keys.
    - `_send`: A function that opens a client socket connects to a server and exchange some information.
    - `_find_keys_for_node`: Return a dict of all keys that belong to new_node_id.
    - `_store_new_value`: Stores a new value locally.
    - `_delete_value`: Deletes a value locally.
    - `_read_value`: Reads a value locally.

Operations

Join

When a new node joins the Chord Distributed Hash Table (DHT), it follows a well-defined process to integrate itself into the existing network. The joining node begins by contacting a known bootstrap node using its host and port. It sends a **JOIN** command to this bootstrap node, requesting information about its successor and predecessor in the ring. Upon receiving the response, the new node updates its successor and predecessor fields accordingly. If the response does not contain valid successor or predecessor information (e.g., in a single-node ring), the node defaults to setting itself as both the successor and predecessor.

Once the successor and predecessor relationships are established, the new node notifies its predecessor and successor about its presence. For nodes using eventual consistency, these notifications are sent asynchronously to minimize latency. A thread is also started in parallel to initiate the process of acquiring keys from its successor using the _acquire_keys function. For nodes using linearizability, these notifications are processed synchronously, ensuring a consistent view of the topology before proceeding with key acquisition.

The _acquire_keys function is invoked to fetch the keys that the new node is now responsible for, based on its position in the ring. The successor node receives a TRANSFER_KEYS command, which includes the ID of the joining node and optionally a time-to-live (TTL) parameter. The successor responds with the keys that fall within the range of the new node's responsibility. These keys are then incorporated into the new node’s data store. This process ensures that the DHT maintains its integrity, with each node responsible for a well-defined portion of the keyspace.

Throughout this operation, the system adapts to the replication consistency model in use. In eventual consistency, keys may propagate lazily to replicas, while in linearizability, stricter guarantees ensure that key replication and updates are immediately consistent. This methodical approach ensures that the DHT remains functional and consistent, even as nodes dynamically join the network.

In summary, when a new node joins the DHT, the system ensures that both the primary keys and their replicas are appropriately shifted to maintain the integrity of the ring and the replication factor. The process begins by notifying the new node’s successor and predecessor of its presence. The new node then acquires its primary keys by sending a `TRANSFER_KEYS` command to its successor, which calculates and moves the keys within the new node's range. Additionally, the successor notifies its own successor to move replicas to their new positions, and this cycle continues for k iterations, ensuring all replicas are correctly redistributed. For instance, in a ring **with nodes 29, 119, and 247**, holding **keys 18, 50, and 70** with a **replication factor of 2**, the addition of a **new node** with **ID 60** results in keys being shifted to maintain both primary and replica positions.

Initially, node 60 acquires keys 18 and 50 from node 119, while node 119 updates its keys by requesting replicas from node 247. This redistribution ends with node 60 storing keys 18 and 50, and node 119 storing keys 50 and 70, demonstrating the efficient propagation of data to maintain consistent replicas across the network. The difference between the 2 levels of consistency is that the one sends the

requests in the background while the chain replication waits for the response of the other node before answering to the initial request.

Put

The `chord_put` method is designed to handle the insertion and replication of data in the Chord DHT. The replication process relies on a **time-to-live (TTL)** mechanism to determine whether the current operation involves the primary node or a replica. If a TTL is present in the request, it indicates that the request has already passed through the primary node, and the current node is a replica. In this case, the node simply stores the data locally, decrements the TTL, and forwards the request to its successor. This ensures that the data is replicated across the network up to the specified replication factor.

When the TTL is not present, the node determines whether it is the primary node for the data by checking if it is responsible for the key (based on the Chord hash). If the node is the primary, it stores the data locally and initiates the replication process by introducing a TTL equal to `self.replication_factor - 1`. The request is then forwarded to the successor to continue replication. If the node is not the primary, it simply forwards the request to the appropriate successor. This process guarantees that the data is stored at the primary node and replicated to the appropriate number of replicas in the network.

The replication behavior differs between **linearizability (chain replication)** and **eventual consistency**. In the case of eventual consistency, replication is handled asynchronously, with no confirmation required from the replicas, allowing the operation to proceed in the background. This improves performance by reducing response time but may result in temporary inconsistencies. On the other hand, for linearizability, replication is performed synchronously, requiring confirmation from each replica before the operation is considered complete. This approach ensures strong consistency, but at the cost of increased latency.

Delete

The same mechanism applies to the `DELETE` operation. When a `DELETE` request is received, the node checks the presence of a TTL. If a TTL is present, the node deletes the specified data, decrements the TTL, and forwards the request to its successor. If the TTL is not present, the node determines whether it is the primary node for the data. If so, it removes the data locally and initiates the process of notifying replicas by setting a TTL. The operation then propagates through the ring to ensure all replicas are updated. The difference in behavior between eventual consistency and linearizability remains consistent, with eventual consistency allowing asynchronous propagation and linearizability enforcing synchronous updates. This design ensures both operations function efficiently within the constraints of the selected consistency model.

Get

The `chord_get` method is designed to retrieve data from the Chord DHT, accounting for both replication and consistency models. When handling eventual consistency, the method prioritizes efficiency by checking if the requested data is available locally. If the node contains the information, it immediately returns the value without querying other nodes, even if it is the primary node. This

approach ensures quick responses but introduces the possibility of reading stale data, as replicas may not yet have been updated with the latest changes.

In contrast, with chain replication (linearizability), the method ensures that the read operation retrieves the most recent and consistent data by querying the tail of the replication chain. Regardless of whether the current node has the information, it forwards the request through the chain to ensure that all replicas maintain the same value. This guarantees strong consistency at the cost of increased latency, as the operation requires traversing the chain to reach the designated node. Both approaches effectively balance the trade-offs between speed and data consistency, aligning with the requirements of their respective consistency models.

Depart

The `depart` process in the Chord DHT ensures the graceful removal of a node while maintaining the integrity of the ring and preserving data consistency. The process begins with the departing node deleting the songs it originally uploaded, ensuring these keys are no longer associated with it. Next, the node notifies its successor and predecessor to update their respective references, effectively linking them together and bypassing the departing node. Following this, the departing node transfers all its keys to its successor using the `MOVE_ALL_KEYS` command. The successor merges the received keys with its own data store and, in turn, forwards the remaining keys to its successor, continuing this cycle up to the replication factor.

At each step, the nodes take into account the keys they are responsible for, ensuring they retain their own primary data while merging the incoming data. This approach ensures that both primary keys and replicas are redistributed correctly across the network. For replication, the process ensures that all nodes within the replication chain receive the necessary updates to maintain the desired replication factor. By the end of this process, the departing node clears its data store and exits the ring, leaving the network in a consistent and functional state.

Experiments

This experiment evaluates the performance of the Chord DHT under different configurations of replication factor K and consistency models (Linearization and Eventual Consistency). The tests involve running insertion and query operations on 10 nodes with K=1, K=3, and K=5. The throughput, measured in requests per second, is calculated for each configuration to compare the performance impacts of varying replication levels and consistency guarantees.

Insert & Query Experiment

Table 1: Throughput Data by Consistency and K

Consistency	K=1 Insert Throughput (req/s)	K=3 Insert Throughput (req/s)	K=5 Insert Throughput (req/s)	K=1 Query Throughput (req/s)	K=3 Query Throughput (req/s)	K=5 Query Throughput (req/s)
Linearization	181.87	178.11	168.47	182.58	176.47	171.10
Eventual Consistency	188.10	181.85	169.77	203.38	207.00	183.99

Table 2: Throughput Summary by K

K	Insert Throughput (Linearization)	Insert Throughput (Eventual Consistency)	Query Throughput (Linearization)	Query Throughput (Eventual Consistency)
1	181.87	188.10	182.58	203.38
3	178.11	181.85	176.47	207.00
5	168.47	169.77	171.10	183.99

Observations and Results

- **Insertion Throughput:**
  The throughput for insertions decreases as K increases for both consistency models. This is expected for chain replication (Linearization) due to the additional overhead of synchronizing across more nodes. For eventual consistency, although updates occur asynchronously, the increased number of background processes reduces individual operation speed, leading to slower overall throughput as K increases.
- **Query Throughput:**
  Queries are faster under eventual consistency, especially as K increases, because more nodes contain the requested data, reducing the need for additional hops. Conversely, for Linearization, the query must always access the tail of the replication chain, resulting in slower throughput as K increases.
- **Replication Factor K=1:**
  When K=1, there is no replication, so both consistency models exhibit similar performance for insertions and queries, as no extra overhead for maintaining replicas exists.

These results highlight the trade-offs between consistency guarantees and system performance, with eventual consistency favoring speed and Linearization prioritizing strong data consistency.

Requests Experiments

$ ./run_requests.sh "l"

Running mixed insert/query requests (k=3)...

All mixed requests completed in 3106836321 nanoseconds (3.106836321 seconds).

Throughput: 160.935417363 requests/second.

$ ./run_requests.sh "e"

Running mixed insert/query requests (k=3)...

All mixed requests completed in 2734822658 nanoseconds (2.734822658 seconds).

Throughput: 182.827211313 requests/second.

Table: Comparison of Linearization and Eventual Consistency

Command	Linearization	Eventual Consistency
query, Hey Jude	{'id': -1, 'value': []}	{'id': -1, 'value': []}
insert, Respect, 528	{'status': 'OK'}	{'status': 'OK'}
query, Hey Jude	{'id': 9, 'value': [' 524']}	{'id': 247, 'value': [' 524']}
query, Like a Rolling Stone	{'id': 9, 'value': [' 583']}	{'id': 247, 'value': [' 583']}
query, Respect	{'id': 9, 'value': [' 528', ' 553']}	{'id': 247, 'value': [' 528', ' 553']}
insert, Respect, 522	{'status': 'OK'}	{'status': 'OK'}
insert, Hey Jude, 542	{'status': 'OK'}	{'status': 'OK'}
query, Hey Jude	{'id': 9, 'value': [' 539', ' 542', ' 523', ' 524', ' 561']}	{'id': 247, 'value': [' 542', ' 561', ' 539', ' 524', ' 523']}

Explanation of Results

1. **Linearization:**
    - All operations are processed in a sequential manner with chain replication. The queries return the latest and most consistent values as they always query the final replica in the chain, ensuring data accuracy.
    - For instance, in the final query for "Hey Jude," the node consistently returns all updated values in a well-defined order.
2. **Eventual Consistency:**
    - Queries return the first available value found, which may not always be the latest. This approach achieves faster throughput as the request doesn't need to traverse the entire replication chain.
    - For example, the final query for "Hey Jude" returns a set of values where the order and content slightly differ from the results in linearization.

Observations

- **Performance and Consistency Trade-offs:**

    - Linearization ensures strong consistency, with queries always returning the latest values. However, this comes at the cost of increased latency, as operations must traverse the chain and wait for responses from all replicas.
    - Eventual consistency prioritizes performance, achieving higher throughput. However, it may return stale data or incomplete results, particularly if updates have not yet propagated to all replicas.
- **Replication Impact:**
    - With eventual consistency, data is stored and retrieved faster, but the consistency of the network depends on the propagation of updates. This allows faster reads, as seen in the varied order and values returned for the same queries.
- **Practical Use Cases:**
    - Linearization is ideal for scenarios requiring strong guarantees about data accuracy, such as financial transactions or inventory systems.
    - Eventual consistency is better suited for applications where performance and availability take precedence over immediate consistency, such as caching systems or social media feeds.