Report 5: Chordy

Alexandru Matcov

October 9, 2024

1 Introduction

This seminar covers the implementations of peer-to-peer (P2P) distributed has table (DHT) according to Chord - a distributed lookup protocol. Data storage can be effectively implemented over Chord by linking each data item with a specific key and saving the key-data pair at the node assigned to that key. Chord adjusts smoothly as nodes enter or exit the network and remains capable of responding to queries even in a dynamic system.

2 Main Problems and Solution

In distributed systems, a key challenge is maintaining a consistent and stable network structure that adapts dynamically as nodes join and leave. The fundamental issue is not only how to manage the nodes but also how to efficiently store and retrieve data within the system. In the context of a DHT, the solution must account for the continuous changes to the ring topology while ensuring that data is correctly distributed and accessible.

2.1 Ring Formation and Stabilization

The first problem arises during the formation and maintenance of the ring. Each node in the DHT has to correctly identify its successor and predecessor to form a stable ring structure. Initially, without data storage capabilities, the key challenge is to maintain the integrity of the ring, especially in a dynamic environment where nodes frequently join or leave.

In a simple case, such as when there is only one node, the node is its own predecessor and successor. As nodes are added, each node must stabilize its position in the ring by periodically querying its successor about its predecessor. This is known as the stabilization procedure, and it ensures that nodes maintain an accurate view of the ring. A frequent stabilizing operation (e.g., every 100 milliseconds) ensures rapid detection of node additions or failures, allowing the ring to quickly self-repair. However, this approach can introduce significant network overhead due to constant traffic. Therefore, the frequency of this operation must balance the trade-off between network efficiency and responsiveness.

For instance, during the stabilization phase, if node N adds a new predecessor P, N must update its keys accordingly and split its data store to transfer the relevant portion of the key space to P. This ensures that both the ring structure and data distribution are consistent.

2.2 Data Storage and Retrieval

Once the ring is correctly stabilized, the next challenge is to implement an efficient data storage and retrieval mechanism. Each node in the ring maintains a local store, typically in the form of a key-value pair. The tricky part comes when a new node joins the system and disrupts the existing key space. When a new node is added, it must take responsibility for part of the key-value store that previously belonged to its predecessor. This involves splitting the store and reassigning the key ranges.

In the solution, each node is responsible for storing data corresponding to its range of keys. If a query for data arrives at a node, and that node is responsible for the key in question, it stores or retrieves the data directly. If not, the node forwards the query to its successor until it reaches the correct node. This guarantees that every key can eventually be located, even as nodes are added or removed.

For example, in one implementation, when a node N identifies a new predecessor P, it keeps the key-value pairs in the range $(P_{key}, N_{key}]$ and transfers the remaining data to P. This mechanism allows the system to adapt dynamically as the ring grows or shrinks, ensuring that each node's data store reflects its current position in the ring.

2.3 Failure Handling

To handle node failures, we implemented a mechanism using Erlang's built-in `monitor/2` procedure. This allows both the predecessor and the successor of a node to detect failures. By extending the node representation to a tuple {`Key, Ref, Pid`}, where `Ref` is the monitor reference, we can track the health of both the predecessor and successor. When a failure occurs, the system receives a `'DOWN'` message, and we invoke a handler that compares the reference from the message with the saved references of the predecessor and successor.

If a successor fails, the ring is repaired by adopting the next node (i.e., the successor's successor). The node then updates its successor, monitors the new one, and continues the stabilization process. If the predecessor fails, the system sets the predecessor to `nil`, waiting for a new node to take its place. This simple failure recovery scheme makes the system fault-tolerant against single node crashes, though it does not protect against multiple consecutive node failures.

To address potential false positives (temporary unavailability), the system relies on the stabilizing procedure to repair the ring when nodes reconnect, reducing the risk of permanent disconnection from the ring. However, this does leave a small window where a node might falsely assume its neighbors are dead.

2.4 Replication Strategy

In addition to failure detection, we implemented a simple replication mechanism to maintain data consistency in the case of node failures. When a node adds a key-value pair to its store, it forwards a {`replicate, Key, Value`} message to its successor, which stores the data in a separate `Replica` store. This ensures that if a node crashes, its successor can merge the replica with its own store and take over the responsibility for the lost data.

When a new node joins the ring, it not only inherits a portion of the store but also a portion of the replica, ensuring that data redundancy remains intact. To avoid inconsistencies, we delay confirming the addition of a key until both the store and its replica are updated. This reduces the risk of lost or duplicated data in the event of a crash during replication.

This solution, while simple, introduces challenges such as handling duplicates when a confirmation is lost or when a node fails between adding the value and replicating it. In practice, a client may need to resend the data if no confirmation is received, and duplicate entries should be managed correctly by the system. Although the system is not foolproof, it provides a reasonable balance between simplicity and fault tolerance for single node failures.

2.5 Evaluation

To evaluate the performance of the DHT, a test module was implemented that launches either 1 or 4 nodes, and 1 or 4 test machines. The nodes manage a set of randomly generated keys, while the test machines perform lookup operations on the keys. For a single test machine, 4000 keys are added and then looked up. In the case of 4 test machines, each adds 1000 keys concurrently, simulating a distributed environment, though these tests are run locally.

Since there is no network latency in these tests, the results may differ from a real distributed system. The lookup times for each configuration are measured in seconds, and multiple test machines' times are aggregated. Table 1 summarizes the average lookup times for different configurations.

The results show that the distributed nature of the DHT greatly improves lookup times when multiple test machines are used on different nodes,

	1 node	4 nodes
1 test machine	3 ms	3 ms
4 test machines on the same node	7 ms	9 ms
4 test machines on different nodes	N/A	2 ms

Table 1: Average lookup time (ms) for 1000 keys

with a significant reduction in average lookup time (2 ms with 4 nodes and 4 test machines). When all test machines are running on the same node, lookup times increase due to resource contention, but the DHT still maintains reasonable performance.

A graph showing the performance improvement with different configurations could be added here to better visualize the advantage of distributing test machines across multiple nodes.

3 Conclusion

In conclusion, the DHT implementation demonstrates efficient scalability and performance, especially when distributed across multiple nodes. The tests showed that distributing both the nodes and test machines significantly reduces lookup time. However, running multiple test machines on the same node leads to performance degradation due to resource contention. The system's ability to adapt and maintain low lookup times highlights its effectiveness in handling key-value storage in a distributed environment.