Distributed Key-Value Store

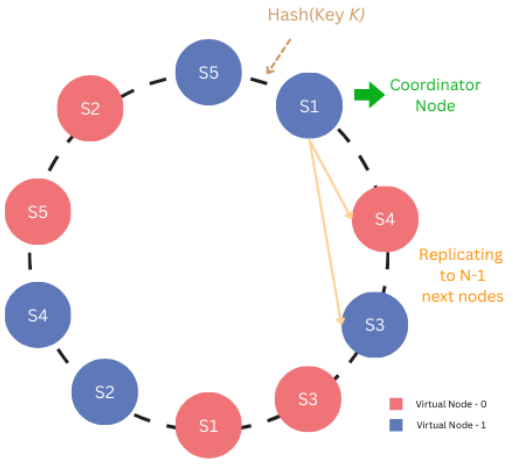
Sahil Das - 21110184 Yashraj J Deshmukh - 21110245

Our implementation draws inspiration from Amazon's Dynamo, employing a consistent hashing system design. This approach aligns with the project's "shared nothing" requirement, as each node operates independently without relying on shared resources. The system prioritizes availability over strict consistency in certain scenarios, achieving eventual consistency through background synchronization processes.

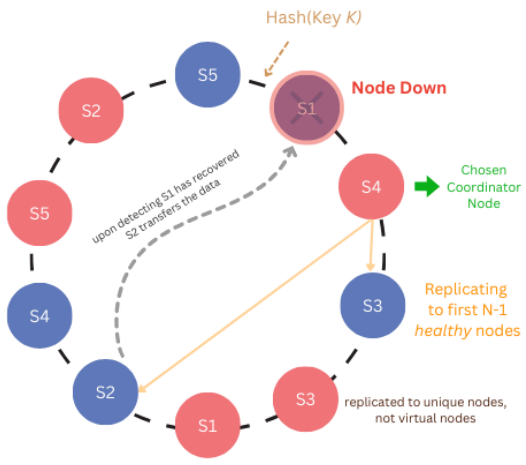
We chose this algorithm to explore a modern protocol not covered in class, as we were interested in learning new concepts and found Dynamo's design intriguing after reading its paper and wanted to deepen our understanding of it. Particularly appealing was its ability to function effectively without excessive data replication across all nodes, offering a balanced approach to distribution and redundancy. This design allows for efficient scaling and fault tolerance without the overhead of replicating all data to every node in the system. Unlike systems that replicate data to all n nodes or require at least n/2 nodes for storage, our approach only replicates to N nodes, where $N < n$. This significantly reduces the replication overhead, especially in larger systems, drastically improving efficiency and scalability.

The system architecture is built around several key components:

- Consistent Hashing: We use a hash ring to distribute keys across nodes. Each node is assigned multiple virtual nodes on the ring, improving load balance. This method could allow for implementation of easy addition or removal of nodes without significant data redistribution.
- Data Partitioning: Keys are mapped to nodes based on their position on the hash ring. This partitioning scheme ensures an even distribution of data and load across the system.
- Replication: To ensure fault tolerance and high availability, we replicate each key-value pair on multiple nodes. The number of replicas (N) is configurable, allowing for a balance between consistency, availability, and partition tolerance as described in the CAP theorem.
- Request Routing: Implemented as an intermediate load balancer, this component determines which node should handle each request based on the key's position on the hash ring. As noted in section 6.4 of the Dynamo paper, this functionality can be considered part of the client-side implementation, acting as a request router. This approach distributes requests efficiently across the system.
- Hinted Handoff: When a node is temporarily unavailable, requests for its keys are routed to the next available node on the ring. This node stores the data with a "hint" of where it should eventually be stored. When the original node comes back online, the data is transferred back, ensuring no loss of writes during node failures. Once the transfer succeeds, the hinted node may delete the object from its local store without decreasing the total number of replicas in the system.



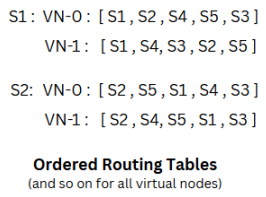
Partitioning and replication of keys in Dynamo ring



Hinted Handoff

- Data Persistence: To handle node crashes and restarts, we implement a simple but effective persistence mechanism using local storage writes. While Dynamo uses more complex structures like Merkle trees for anti-entropy and synchronization, our approach balances the need for crash recovery with the project's scope. Each node periodically writes its data to disk, allowing for recovery after a crash. SQL could have been an alternative for persistent storage, but our chosen method adequately serves the project's requirements.
- Eventual Consistency: Our system favors availability over strong consistency in certain scenarios. This means that during network partitions or node failures, the system continues to accept reads and writes. Conflicting versions are reconciled later ensuring eventual consistency across all replicas.
- Quorum-based Reads and Writes: We implement configurable read (R) and write (W) quorums. A write operation must be acknowledged by W replicas to be considered successful, while a read operation must query at least R replicas. By tuning these parameters, we can adjust the balance between consistency and availability.

The request router plays a crucial role by interfacing between clients and the distributed storage nodes. It maintains the consistent hash ring, preference list for every node, and determines the coordinator node for each key in a request. This component ensures that clients can interact with the system as if it were a single entity, abstracting away the complexities of the distributed architecture.



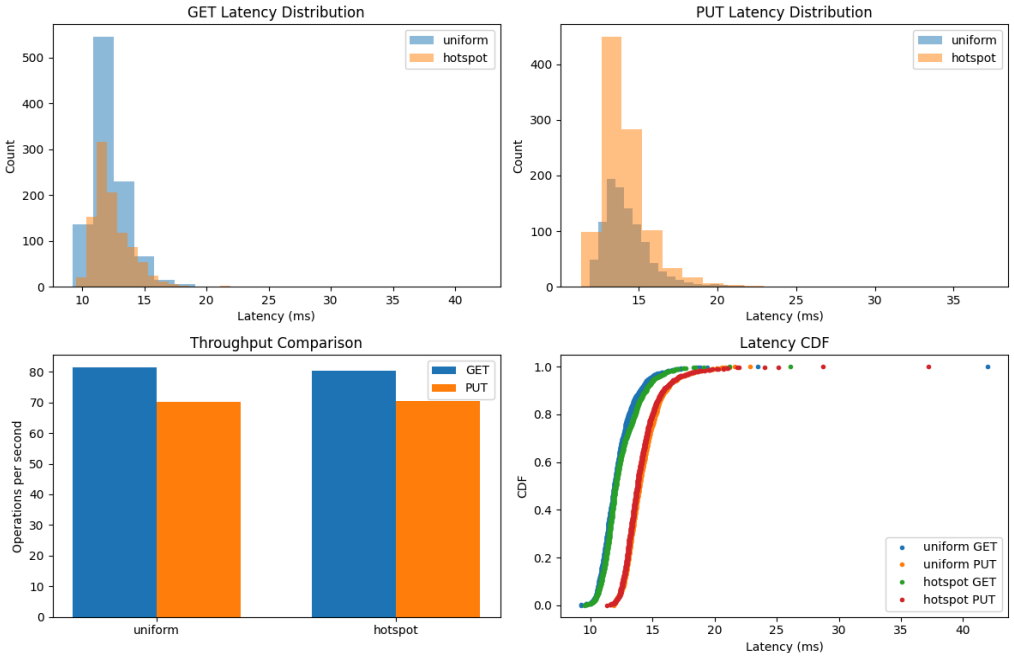
The system aims for eventual consistency, allowing temporary inconsistencies during network partitions or node failures. To address potential conflicts, a timestamp-based reconciliation approach can be used. When conflicting versions are detected, timestamps are compared and a "last write wins" strategy is applied, preserving the most recent update (This is used in Cassandra as well). For future enhancements, we have thought up a Version Control Number (VCN) system. This approach would assign a unique, incrementing number to each update, providing a more precise method for tracking and reconciling changes across the distributed system.
Considering the replication factor N, the system requires at least $\lceil n/N \rceil$ nodes to remain active for complete database availability, where n is the total number of nodes. This is because every N-th node on the ring stores data for its $N-1$ predecessors. Thus, in the *best* worst-case scenario, the system can tolerate up to $n-\lceil n/N \rceil$ node failures. For example, with N=3, the system can handle failures of up to 67% of nodes, and this tolerance increases with higher N (e.g., 75% for N=4, 80% for N=5). However, if consecutive N unique nodes on the ring fail, there would be a loss of availability for approximately 1/n of the data, assuming a uniform distribution of key hashes.
Dynamo is built like an ‘always writable’ data-store as long as just W(<<n) are up, including one node from the preference list. Timestamp-based reconciliation could further enhance the system’s availability and eventual consistency. Even if some nodes are temporarily unavailable due to failures, writes can still proceed using hinted handoff, and reads can fetch data from any available replica. The reconciliation process would ensure that any inconsistencies caused by concurrent updates or node failures are eventually resolved.

Our implementation strives to balance the project requirements with the principles of distributed systems, offering a robust and scalable key-value store solution. By adopting key concepts from Dynamo while simplifying certain aspects, we've created a system that demonstrates the core principles of distributed key-value stores while remaining within the scope of the project.

Reference: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
Github Repo: https://github.com/rae-1/Distributed-KV-Store (refer to README and correctness tess here)

We have built our distributed key-value store in Python, which does not natively support generating .so files. Instead, our implementation focuses on providing the required functionality through Python modules.

Performance Test:



In uniform distribution, all keys are accessed with equal probability. Thereby simulating a balanced workload where every piece of data is equally important. Whereas in Hotspot Distribution a small subset of keys are accessed much more frequently. Hence, simulating a real-world scenarios where certain data is popular.

1. On average, GET operations take 12.2-12.4ms and are consistently faster than PUT operations which take around 14.1-14.2ms. This lower value can be attributed to the functioning of both operations. In GET, we don't have to write to persistence storage, but in PUT it is necessary. Further, there is an overhead of acquiring locks in PUT, because we implemented it in an asynchronous manner as we need to wait for the first W number of acknowledgments.
2. Moreover, the GET operations beat PUT in terms of throughput as well. It shows that the system is well optimized for read operations, and also the hotspot pattern performs almost identically to uniform. This indicates that we are able to achieve effective load balancing.
3. The 99%tile in all the cases is close to the average latency suggesting that consistent performance with few outliers.

Correctness Test:

The distributed key-value store testing framework provides comprehensive correctness guarantees through systematic validation of critical system properties. By executing a sequence of operations that verify data integrity during normal operations, followed by targeted server failures that challenge the system's fault tolerance mechanisms, the tests confirm the store's ability to maintain consistency even when preferred nodes are unavailable. The framework specifically validates that write and read quorums function correctly during partial system failures, and that the hinted handoff mechanism properly facilitates data synchronization when failed servers recover. Through this methodical approach, the tests confirm the system correctly implements eventual consistency while maintaining availability during network partitions, thus validating the fundamental theoretical guarantees of a distributed key-value store in practice.