Implementation and Analysis of a Weakly Consistent Key-Value Store - Dynamo

Anirudh Garg (ag9563), Saiteja Siddana (ss14789)

Spring 2024

1 Introduction

In this project, we embarked on implementing and analyzing a weakly consistent key-value store, drawing inspiration from concepts discussed in our class. Our main focus was on implementing a version of Dynamo, a key-value store developed by Amazon, and then analyzing its behavior under varying conditions of failure rates, message latencies, and message drop rates. The primary objective of this project was to gain insights into the impact of weak consistency on data integrity and system performance.

The goal of implementing and analyzing a weakly consistent key-value store is particularly intriguing due to its real-world relevance in modern distributed systems, where balancing availability and partition tolerance is crucial. Delving into advanced distributed system concepts like anti-entropy mechanisms and gossip protocols offered us an opportunity to explore cutting-edge technologies and gain hands-on experience with fault-tolerant architectures. The project's analytical focus, including testing strategies and performance metric measurement, contributed to a rigorous understanding of weak consistency's impact, aligning well with our interests in distributed systems and software engineering.

2 Architecture

2.1 Dynamo Module

The `Dynamo` module is the core of the system, encompassing all functionalities related to node management, data storage, and communication. Key components of the module include:

- **State Management**: Each node maintains a state defined by the `Dynamo` struct, which includes:
  - `view`: A map indicating the availability of other nodes.
  - `hash_map`: A key-value store residing on the node.
  - `node_list`: An ordered list of nodes within the system.
  - `key_range_data`: Specifies the range of keys that each node is responsible for.
  - `gossip_timer` and `gossip_timeout`: Used for managing the gossip protocol timing for consistency.
  - `merkle_tree_map`: Potentially for future use in optimized synchronization.
- **Configuration and Initialization**: Nodes are initialized with a `new_configuration` function, setting up initial parameters such as node lists and key ranges.

2.2 Key Functionalities

- **Data Operations**: Includes basic **put** and **get** functionalities allowing for writing and reading data from the key-value store. The system checks if the node is responsible for the key and redirects the request if necessary.
  - **Put Operation**: `put(key, value, context)`

  - **Get Operation**: `get(key)`

- **Gossip Protocol**: Utilized for maintaining consistency and synchronization across nodes. It handles updates and propagates them through the network to keep all nodes' views consistent.
- **Failure Handling**: Nodes periodically check the liveness of other nodes using the gossip protocol and mark nodes as dead or alive based on timeouts and responses received.
- **Read and Write Quorums**: The system ensures eventual consistency through configurable read and write quorums, determining how many nodes need to respond to read and write operations respectively.

2.3 Utility Functions

- **Node and Process Management**: Functions like `checkProcessAlive` and `fetchAliveList` are crucial for determining the operational status of nodes.
- **Key Range Management**: Functions such as `isValidRange` and `check_valid_range` ensure that data requests are directed to the correct nodes responsible for the key ranges.
- **Gossip and Synchronization**: Functions like `reconcile_views` and `reconcile_all_failed_process` manage the synchronization of node states across the cluster.

2.4 Client Module

The `Dynamo.Client` module provides an interface for external clients to interact with the Dynamo cluster. It supports operations like:

- **Put and Get Requests**: Clients can store and retrieve data by interacting directly with the nodes managing the relevant key ranges.
- **Process Management**: Clients can trigger nodes to stop, facilitating tests of the system's fault tolerance.

2.5 Challenges and Observations

- **Concurrency and Synchronization**: Managing state consistency in a distributed environment without centralized control is challenging and requires careful implementation of synchronization protocols like gossip.
- **Fault Tolerance**: The dynamic nature of node availability necessitates robust mechanisms for failure detection and recovery, which are addressed through the gossip protocol and periodic checks.
- **Scalability**: As the number of nodes and the volume of data increase, maintaining performance and efficient data distribution becomes more complex.

3 Analysis Metrics

By incorporating the following analytics metrics into our project, we aim to conduct a comprehensive evaluation of weak consistency in key-value stores and gain actionable insights into its implications for distributed system design and performance.

- **Performance metrics**:
  - **Latency**: We measure the time taken for read and write operations to complete for different message delay values and (R,W) values.
  - **Throughput**: We assess the system's throughput by measuring the rate of successful read and write operations per unit of time for different message delay values and (R,W) values.
- **Frequency of stale data reads**: We propose a measurement approach based on tracking read operations returning stale data through timestamp or version number comparison, aiming to analyze the impact of weak consistency on data accuracy and client experience.

4 Testing Strategy

To ensure the thoroughness and reliability of our analysis, we will employ a comprehensive testing strategy that encompasses various aspects of system behavior under weak consistency. Our testing strategy will include the following components:

- **Simulated Environment Setup**: We will create a simulated distributed environment using the `Emulation` module. This module provides functionality to simulate network conditions, introduce failures, and vary message latencies and drop rates within the code itself. We utilize this simulated environment to test our implementation's behavior under different scenarios directly within the codebase.
- **Scalability Testing**: We will evaluate the scalability of our weakly consistent key-value store by deploying it on different numbers of nodes. Starting from a small cluster, we will gradually increase the number of nodes to assess how the system handles increased load and data replication across nodes. Scalability testing will help us understand the performance issues of weak consistency as the system scales.
- **Message Delay Simulation**: To simulate real-world network conditions, we will intentionally introduce message delays between nodes at varying rates. This approach enables us to assess the impact of delayed messages on data consistency and system behavior.
- **Process Failures Performance Testing**: This involves analysing the latency of the system in different process failure scenarios, namely, 1 or 2 process fail with different (R, W) values.
- **Message Drop Simulation**: To simulate real-world network conditions, we will intentionally drop messages between nodes at different rates. This will allow us to evaluate the impact of message drops on data consistency.

5 Results

All the following results have been obtained with N = 5 (with N being the replication factor) and total number of nodes = 10.

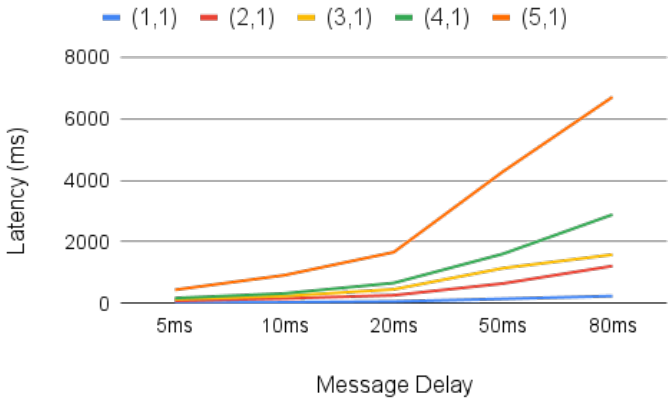

Figure 1: Read Latency

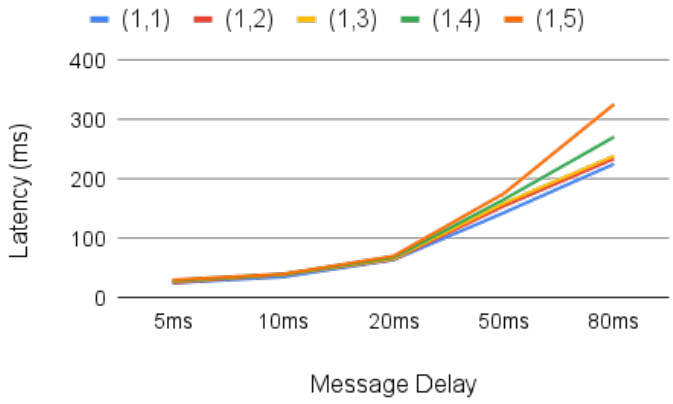

Figure 2: Write Latency

The testing reveals a significant correlation between read/write latency and message delay rates. As we transitioned from sloppy to strict quorums and increased message delay rates from 5 ms to 80 ms, we observed a noticeable rise in the average latencies for both read and write operations as observed in Figure 1 and Figure 2. This observation underscores the critical influence of quorum configurations and message delay rates on operational latencies within distributed systems. Specifically, stricter quorum settings and higher message delay rates tend to amplify latency, emphasizing the importance of carefully tuning these parameters for optimal system performance and data consistency in distributed environments.

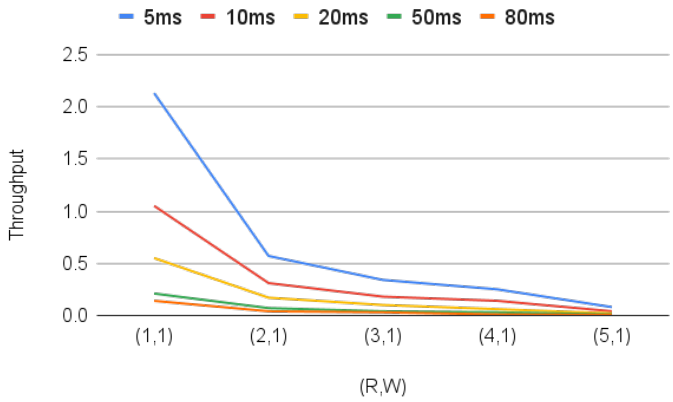

Figure 3: Throughput with Message Delay

The observed increase in latencies, resulting from the transition from sloppy to strict quorums and the introduction of higher message delay rates, has a direct impact on the throughput of the system. Throughput, in the context of a system's performance, refers to the rate at which it can process a certain volume of work or data within a given time frame. As latencies rise due to these factors, the time taken to complete individual read and write operations also increases. This prolonged processing time per operation effectively reduces the overall number of operations that the system can handle within the same time frame. Consequently, the system's throughput experiences a noticeable decrease since it can process fewer operations per unit of time as observed in Figure 3.

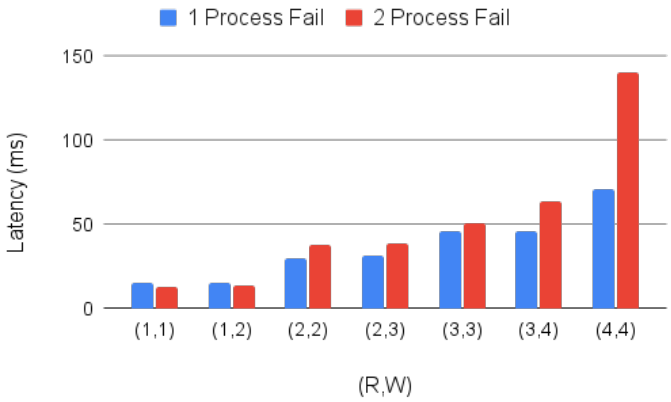

Figure 4: Read Latency with Process Failure

The impact of node failures on system performance was evident in our observations, particularly when dealing with 1 and 2 node failures. During these scenarios, we noted a significant increase in the average read latency, which surged from a baseline of 14 ms to approximately 140 ms as shown in Figure 4. This substantial escalation in latency can primarily be attributed to two key factors: the adoption of expanded read quorums and the requisite view reconciliation procedures triggered by the occurrence of node failures.

- **Expanded Read Quorums:** The utilization of expanded read quorums, especially in the presence of node failures, necessitates additional coordination and agreement among a larger subset of nodes before proceeding with read operations. This expanded coordination inherently introduces higher latencies as the system must ensure that a sufficient number of nodes acknowledge and confirm the validity of read requests.
- **View Reconciliation:** The occurrence of node failures initiates view reconciliation processes within the system. View reconciliation involves updating the system's view of active and available nodes, redistributing responsibilities, and ensuring data consistency in light of the failed nodes. These reconciliation tasks, while crucial for system resilience and data integrity, contribute to the observed increase in read latencies as they involve additional communication and computational overhead.

An interesting thing to note here is that it's important to have robust fault tolerance mechanisms as more process failures result in disproportionately high latencies in case of stricter quorums.

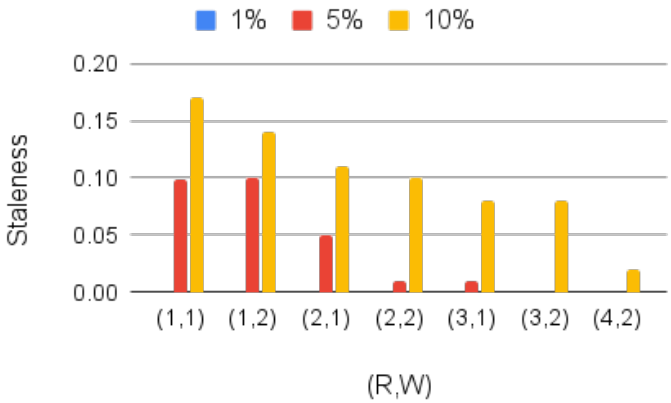

Figure 5: Staleness with Message Drop

For testing staleness with message drop, we sent 10 put requests and after each Put request, we sent

a Get request to each of the 5 (=N) nodes in the preference list. This was done in the scenario with no process failures. We found that upon transitioning from sloppy to strict quorums significantly improves data consistency, particularly in environments with varying drop rates such as 1%, 5%, and 10%. This shift is crucial as it guarantees the selection of the most recent value from the majority of replicas, thereby reinforcing the critical role of quorum configurations in maintaining data integrity within distributed systems. Notably, as drop rates increase, the percentage drop in staleness diminishes when transitioning from lower to higher quorums. This observation underscores the effectiveness of strict quorums in reducing data staleness and highlights their ability to mitigate the impact of higher drop rates on data consistency, showcasing the importance of choosing appropriate quorum configurations based on the system's reliability requirements and environmental conditions.

An interesting thing to note here is that for higher drop rates, the percentage drop in staleness is lower when moving from lower to higher quorums.

6 Conclusion

The comprehensive analysis of various scenarios in distributed systems highlights critical insights into system behavior under different conditions. Transitioning from sloppy to strict quorums and increasing message delay rates significantly impact operational latencies, underlining the crucial role of quorum configurations and message delays. This directly influences throughput, where increased latencies lead to reduced system throughput. Moreover, the shift towards strict quorums improves data consistency, particularly evident in mitigating staleness and selecting the latest values amidst varying drop rates. However, under node failures, there is a noticeable surge in read latency due to expanded read quorums and view reconciliation requirements. These findings emphasize the intricate balance between system performance, data consistency, and fault tolerance mechanisms crucial for designing robust distributed systems.

7 Future Work

In future work, implementing an anti-entropy mechanism for replica synchronization using Merkle trees presents a promising avenue for handling permanent process failures in distributed systems. Anti-entropy mechanisms, such as Merkle tree-based synchronization, can enhance data consistency and fault tolerance by periodically comparing and synchronizing data replicas across nodes. By incorporating Merkle trees, which efficiently summarize data changes, the overhead of synchronization can be minimized while ensuring data integrity. This approach can be extended to handle permanent process failures by automatically detecting and recovering from discrepancies between replicas, thereby enhancing system resilience and reducing manual intervention in fault recovery processes. Additionally, exploring techniques to optimize Merkle tree-based synchronization algorithms for large-scale distributed systems and integrating them seamlessly into existing fault-tolerant mechanisms would be a valuable direction for future research and development efforts in distributed computing.

8 References

- Giuseppe, D., Deniz, H., Madan, J., Gunavardhan, K., Avinash, L., Alex, P., Swaminathan, S., Peter, V., and Werner, V.: Dynamo: Amazon's Highly Available Key-value Store. Amazon.com 2007
- Bailis P, Venkataraman S, Franklin M J, Hellerstein J M, Stoica I. Quantifying eventual consistency with PBS. Communications of the ACM, 2014
- Gilbert, S.; Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM Sigact News 2002