Distributed Key-Value Datastore

Problem Statement

Design and implement a distributed key-value store system using Go Lang. The system should consist of multiple server nodes that communicate with each other to store and retrieve key-value pairs.

Non-Functional Requirements

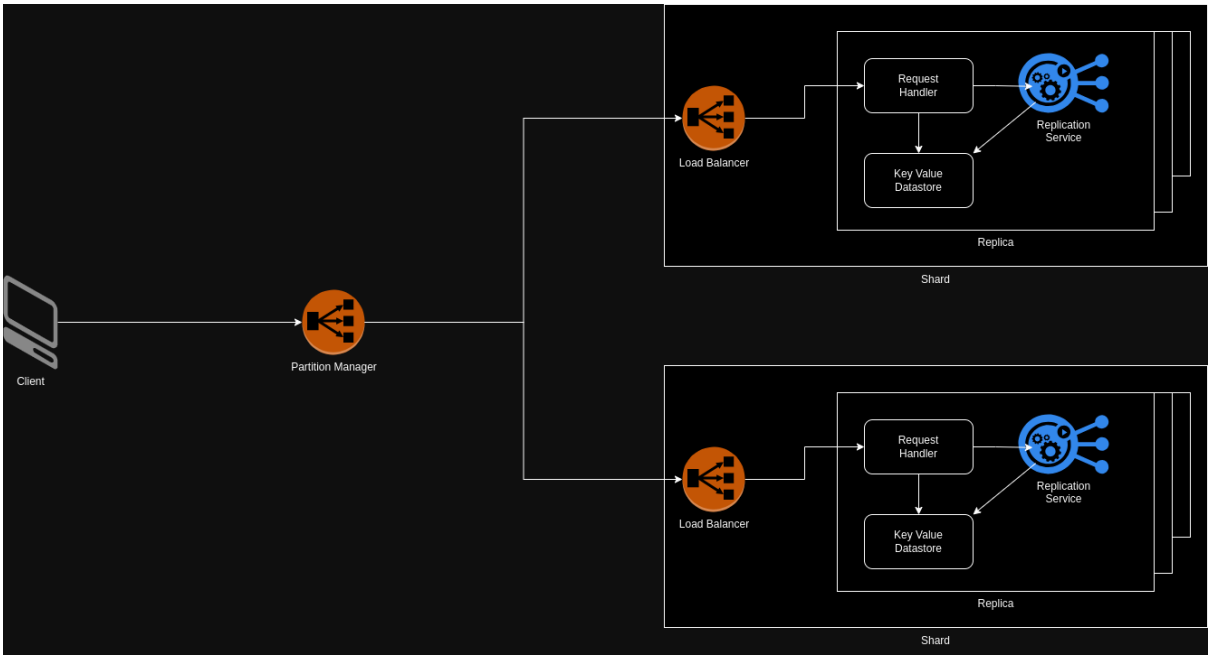
1. **Horizontal Scalability**: The system should be designed to be horizontally scalable, allowing for the addition of new nodes to handle increased load efficiently. This ensures that the system can accommodate growing data storage requirements and increasing workload demands.
2. **Strong Consistency Guarantees**: The system must ensure strong consistency guarantees across all nodes, even in the presence of failures and network partitions. This ensures that all nodes agree on the state of the data and prevents data inconsistencies or conflicts.
3. **Fault Tolerance and Data Integrity**: Mechanisms should be implemented to handle node failures gracefully and maintain data availability and integrity. This includes strategies for data replication, redundancy, and failover to ensure that data remains accessible and consistent under various failure scenarios.
4. **Efficient Concurrent Operations**: The system should support concurrent read and write operations efficiently while maintaining data consistency. This involves implementing concurrency control mechanisms such as locking, versioning, or distributed consensus protocols to prevent data races and ensure correct and reliable operation in a multi-threaded environment.

Functional Requirements

1. **Store Key-Value Pair**: Users should be able to store a key-value pair in the database server.
2. **Retrieve Value by Key:** Users should be able to fetch a value by providing its corresponding key from the database server.
3. **Update Value by Key**: Users should have the capability to update a value associated with a specific key in the database server.
4. **Delete Key-Value Pair**: Users should be able to delete a key-value pair from the database server by specifying the key.

System Design

High-Level Design



Load Balancer and Partition Management

A load balancer has been strategically placed within the system architecture to effectively distribute incoming requests across multiple server nodes. Not only does this balance the workload, but it also assumes the crucial responsibility of partition management. By employing partitioning techniques, the load balancer ensures that the data is distributed evenly among server nodes. This approach prevents any single server node from being overwhelmed with excessive traffic, thereby optimizing system performance and resource utilization.

Data Sharding for Data Integrity and Durability

Data sharding is utilized as a key strategy to enhance data integrity and durability within the Distributed Key-Value Store. By partitioning data across multiple server nodes, we ensure that no single point of failure exists within the system. Each shard is responsible for storing a subset of the overall dataset, thereby distributing the data load and reducing the risk of data loss in the event of a server failure. This approach not only increases the system's scalability but also improves fault tolerance and resilience.

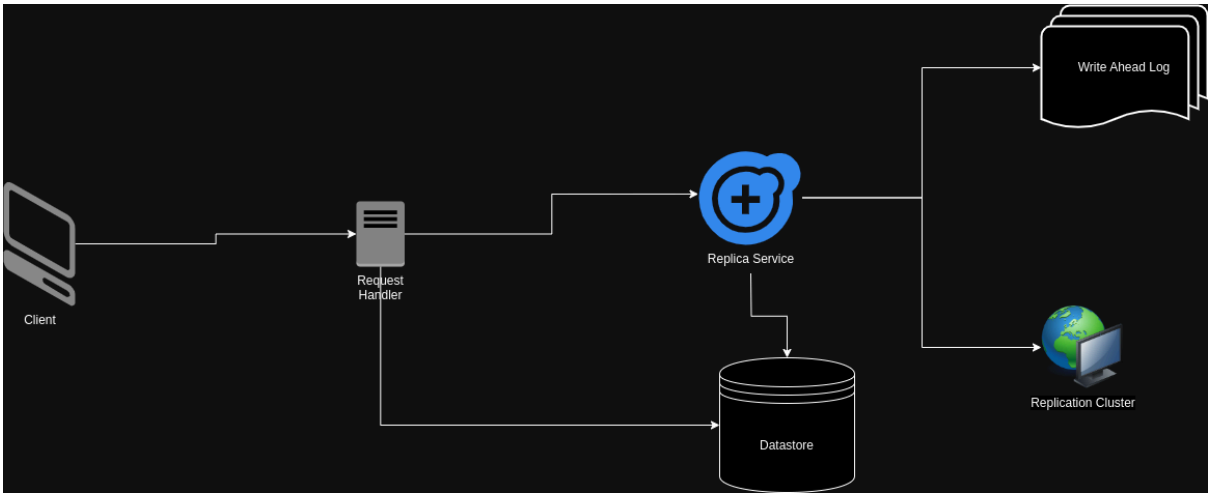
Database Replication Strategy within Shards

Within each shard, multiple server nodes collaborate to implement a robust Database Replication Strategy. This strategy is instrumental in ensuring data availability and maintaining strong consistency across the distributed environment. Through synchronous or asynchronous replication mechanisms, data updates are propagated seamlessly across all nodes within the shard, guaranteeing that all replicas remain synchronized and up-to-date. By adhering to stringent consistency models, such as linearizability or serializability, we enforce data consistency and integrity, thereby enhancing the reliability and trustworthiness of the system.

Benefits of the High-Level Design

- **Scalability**: The use of data sharding and partition management enables the system to scale effortlessly as the dataset grows. New server nodes can be added dynamically to accommodate increased data storage requirements, ensuring that the system remains responsive and efficient even under heavy workloads.
- **Data Integrity and Durability**: Through careful implementation of data sharding and replication strategies, we prioritize data integrity and durability. By distributing data across multiple nodes and replicating it within each shard, we mitigate the risk of data loss and ensure that data remains available and consistent at all times.
- **High Availability and Consistency**: The combination of load balancing, partition management, and database replication ensures high availability and strong consistency within the Distributed Key-Value Store. By distributing workload evenly and synchronizing data updates across replicas, we minimize downtime and eliminate data inconsistencies, thereby enhancing the overall reliability and usability of the system.

Low-Level Design: Replica



Within each server node, several essential components work together to ensure efficient data handling, replication, and fault tolerance:

1. **Request Handler**: This component efficiently manages incoming data requests, ensuring proper routing and processing of key-value operations within the system. Responsible for processing incoming data requests, the request handler ensures that key-value operations are executed efficiently and accurately. It handles tasks such as parsing requests, validating input, and coordinating data access within the in-memory database.

2. **In-Memory Database**: An in-memory database is utilized to store and manage the key-value pairs associated with the distributed key-value store. This database facilitates rapid data access and manipulation, enhancing overall system performance. The in-memory database stores key-value pairs in volatile memory, enabling rapid data access and manipulation. It provides efficient read and write operations, allowing for high-performance data processing within the distributed environment.

3. **Replication Controller**: The replication controller oversees the replication of data across the Replica Cluster Network. It coordinates with other nodes within the cluster to ensure consistency and synchronization of data updates, thereby maintaining data integrity and availability. Tasked with maintaining data consistency and synchronization across the Replica Cluster Network, the replication controller coordinates the replication of data updates among nodes within the cluster. It ensures that changes made to the database are propagated to all replicas in a timely and consistent manner, thereby preventing data inconsistencies and ensuring high availability.

4. **Write-Ahead Log (WAL)**: The write-ahead log serves as a crucial mechanism for ensuring fault tolerance and data durability. It records a log entry for every write operation performed on the database, ensuring that data modifications are persisted even in the event of network failures or system crashes. The write-ahead log serves as a persistent record of all write operations performed on the database. By logging changes before they are applied to the database itself, the WAL ensures that data modifications are recoverable in the event

of system failures or crashes. This mechanism enhances fault tolerance and data durability within the distributed key-value store.

Benefits and Importance

- **Fault Tolerance**: The use of a write-ahead log ensures that data modifications persist even in the face of network failures or system crashes, enhancing the system's fault tolerance and resilience.

- **Consistency and Synchronization**: The replication controller ensures that data updates are propagated consistently across all replicas within the cluster, maintaining data integrity and synchronization across the distributed environment.

- **Performance**: The combination of an in-memory database and efficient request-handling mechanisms optimizes system performance, enabling rapid data access and manipulation for key-value operations.

Sharding Strategy: Range-Based Sharding

Overview

The sharding strategy implemented for distributing data within the Distributed Key-Value Store is a range-based approach. This strategy aims to horizontally distribute the data load across multiple shards by partitioning the dataset into distinct ranges based on the key values.

Implementation Details

Key-Based Sharding

Sharding is performed based on the key provided with each data entry. The sharding process utilizes a hashing function, denoted as H(Key), which calculates the appropriate shard for a given key. The function is defined as follows:

```
H(Key) = AsciiValueOf(LastCharacter(Key)) % ShardCount
```

Hashing Function Explanation

The hashing function H(Key) generates a hash value by computing the modulus of the ASCII value of the last character of the key and the desired shard count. This ensures that each key is consistently mapped to a specific shard within the system.

Benefits

- **Horizontal Load Distribution:** By partitioning the dataset into multiple ranges and distributing them across shards, the range-based sharding strategy facilitates horizontal load distribution. This helps in balancing the workload across the system and improving overall system performance and scalability.
- **Key-Based Deterministic Mapping**: The use of the hashing function ensures that each key is deterministically mapped to a specific shard, providing predictable data distribution and retrieval patterns. This simplifies data management and querying operations within the distributed key-value store.
- **Scalability:** The range-based sharding strategy allows for seamless scalability as the dataset grows. Additional shards can be added to accommodate increased data storage requirements, ensuring that the system remains responsive and efficient under varying workloads.

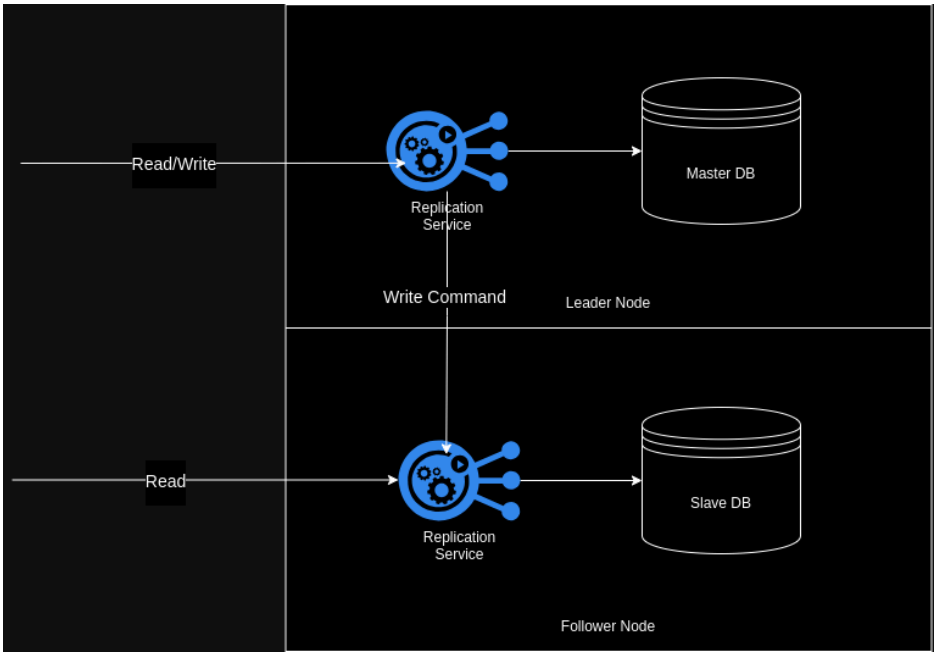
Replication Strategy: Leader-Follower with Raft Algorithm

The replication strategy adopted for ensuring consistent replication of the database within the Distributed Key-Value Store is based on the Leader-Follower model, utilizing the Raft Algorithm. This strategy aims to handle large volumes of data, ensure strong consistency, and effectively distribute read operations across regions.

Implementation Details

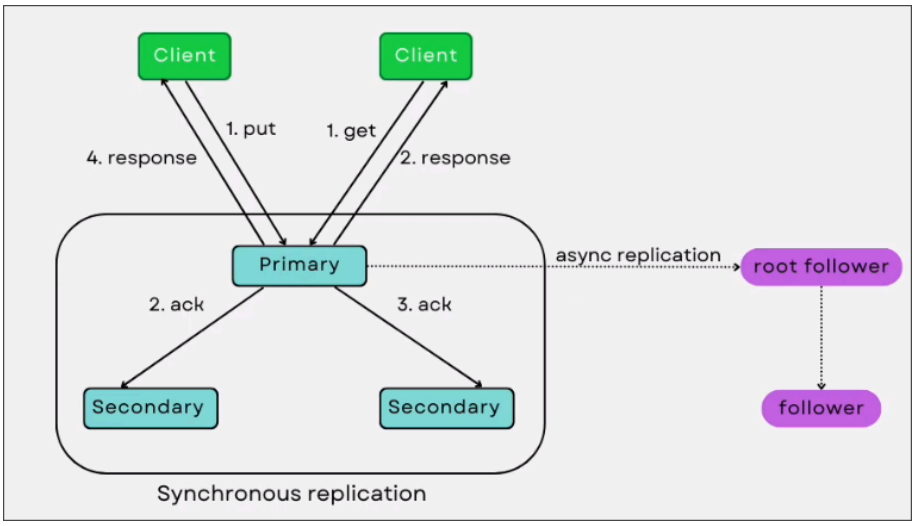
Leader-Follower Model

The Leader-Follower model is employed to establish a consistent replication scheme within the cluster. In this model, a single leader node is elected to manage the replication process, while follower nodes replicate data from the leader. This ensures that all nodes within the cluster maintain synchronized copies of the database, facilitating fault tolerance and data consistency.



Raft Algorithm

The Raft Algorithm is utilized as the underlying consensus mechanism for implementing the Leader-Follower replication strategy. Raft operates on the principle of leader-driven consensus, where a leader node is responsible for managing a replicated log across all nodes in the cluster. The leader coordinates data replication and ensures that all updates are applied consistently across the cluster, thereby achieving strong consistency and fault tolerance.

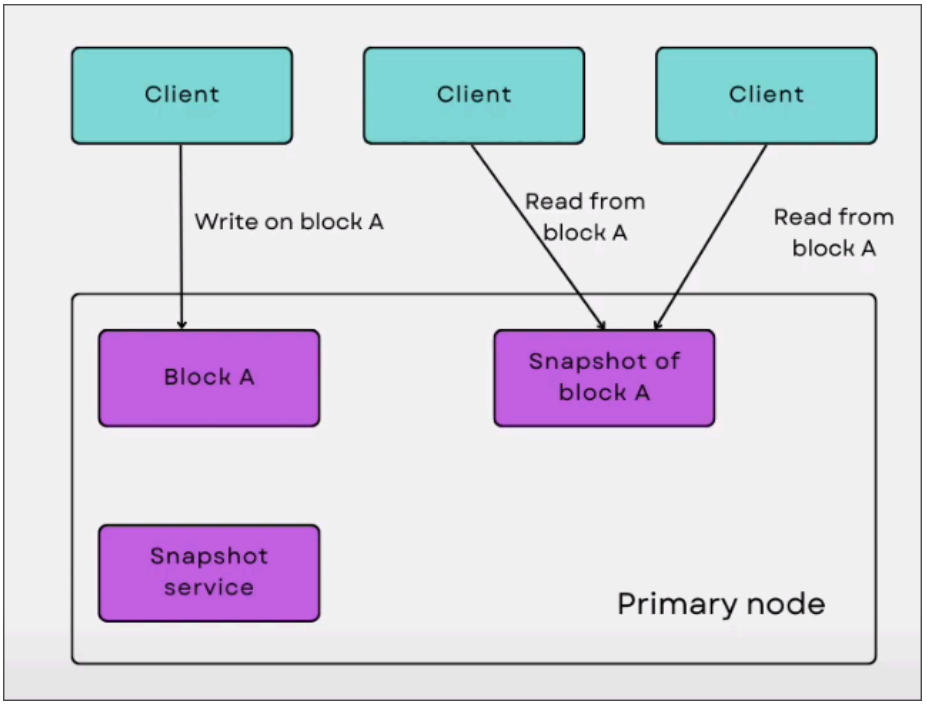


Benefits

- **Strong Consistency**: The Leader-Follower replication strategy, coupled with the Raft Algorithm, ensures strong consistency of data across all nodes within the cluster. The leader's role in coordinating data replication and enforcing consensus guarantees that all replicas remain synchronized and up-to-date.
- **Fault Tolerance**: By electing a leader to manage the replication process, the system achieves fault tolerance by ensuring that data updates can continue to be processed even in the event of node failures or network partitions. The Raft Algorithm's leader election mechanism enables seamless failover and recovery within the cluster.
- **Regional Read Distribution**: The Leader-Follower model facilitates efficient read distribution across regions by allowing follower nodes to handle read requests locally. This minimizes latency and improves performance for read-intensive workloads, enhancing the overall scalability and responsiveness of the system.

Snapshot Strategy

In addition to the Leader-Follower replication model, a Snapshot Strategy is implemented to segregate read and write data within the data store. When writing data into the datastore, a snapshot of the data is created, and read operations are performed from this snapshot. This strategy helps to optimize read performance and minimize the impact of concurrent write operations on read consistency.



The adoption of the Leader-Follower replication strategy, supported by the Raft Algorithm and Snapshot Strategy, provides a robust foundation for ensuring consistent replication, strong consistency, and fault tolerance within the Distributed Key-Value Store. By leveraging these mechanisms, the system achieves high availability, scalability, and performance, making it well-suited for handling large-scale distributed data workloads across regions.

Deployment Instruction:

Requirement:

- Docker
- Docker compose

Steps

- Clone the code from github
- Run `docker compose up -d`

Tech Stack

Programming Language

- Go

Datastore

- In-memory

Logging

- Raft-Bolt-DB by HashiCorp

Load Balancing and Partition Management

- HAProxy

Containerization

- Docker
- Docker Compose

Go Libraries

Web Server Implementation

- gorilla/mux: A powerful HTTP router and URL matcher for building Go web servers.

RAFT Replication Strategy

- raft: A Go library that provides an implementation of the Raft consensus protocol. Used for implementing the RAFT replication strategy for distributed systems.

Summary

Your tech stack and Go libraries are well-organized to support the development of a distributed key-value store. With Go as the programming language, an in-memory datastore, Raft-Bolt-DB for logging, HAProxy for load balancing and partition management, and Docker/Docker Compose for containerization, your system architecture is poised to achieve scalability, fault tolerance, and high availability. Additionally, leveraging gorilla/mux for web server implementation and the raft library by HashiCorp for Raft replication strategy ensures efficient and reliable development of your distributed key-value store.