Term: Fall 2024 **Subject:** Computer Science & Engineering (CSE) **Number:** 512

Course Title: Distributed Database Systems (CSE 512)

Simulating a Redis Distributed Key-Value Cluster

13th October 2024

The Scalable Squad

Deebthik Ravi - 1229479357

Deva Dharshini Ravichandran Lalitha - 1229734859

Santosh Gokul Narayanan - 1228858516

Vikram Kumaresan - 1229638689

1. Introduction

1.1. Background

Distributed database systems have emerged as a critical component in modern computing architectures, driven by the increasing demand for scalable, high-performance data management solutions. These systems consist of multiple interconnected nodes that work collaboratively to store, manage, and retrieve data. Unlike traditional centralized databases, distributed database systems provide enhanced availability and resilience by distributing data across various geographic locations and hardware infrastructures. This architecture is essential for applications that require real-time data access, high transaction throughput, and the ability to handle large volumes of data, such as e-commerce, social media, and IoT applications.

In the context of distributed key-value systems, these specialized databases further streamline data storage and retrieval by employing a simple key-value pair model. This model allows for high-speed read and write operations, making distributed key-value stores particularly suitable for applications that prioritize performance and scalability. They enable horizontal scaling by allowing additional nodes to be added seamlessly, thus accommodating growing data volumes. The distributed nature of key-value stores also ensures fault tolerance, maintaining data availability even during hardware failures.

Distributed key-value stores represent a significant advancement in the realm of distributed database systems, providing a balance of performance, scalability, and reliability. Their simplistic design allows developers to store data as pairs of keys and values, facilitating efficient data retrieval and manipulation. This simplicity leads to faster operations compared to more complex relational databases, making key-value stores ideal for applications with high transaction rates.

Moreover, distributed key-value clusters, such as Redis, incorporate essential features like replication, partitioning, and varying consistency models. These features empower developers to optimize their applications based on specific performance and reliability needs. For instance, replication ensures data durability and availability even in the event

of node failures, while partitioning allows for balanced data distribution across nodes, improving load management and access speed. Additionally, the flexibility in consistency models enables developers to choose between eventual and strong consistency, tailoring their approach to meet application-specific requirements.

Research in distributed database systems has highlighted several foundational principles and challenges associated with distributed key-value stores. The CAP Theorem, proposed by Eric Brewer, asserts that in distributed systems, it is impossible to simultaneously guarantee consistency, availability, and partition tolerance. The work of T. J. Watson provides insights into how different consistency models (eventual and strong consistency) impact the performance of distributed systems.

Several existing solutions have addressed the needs of online gaming applications through distributed key-value stores:

1. **Redis**: As a leading in-memory key-value store, Redis offers high performance and supports complex data structures. Its clustering capabilities enable horizontal scalability, making it suitable for applications requiring rapid data access
2. **Cassandra**: This NoSQL database prioritizes availability and partition tolerance while allowing tunable consistency. Its distributed architecture supports high write and read throughput
3. **Amazon DynamoDB**: This fully managed NoSQL database service provides automatic scaling and high availability. DynamoDB's eventual consistency model allows for low-latency data access

Since our project is trying to simulate an existing key-value store cluster system i.e. It wouldn't necessarily be addressing gaps but it would certainly be replicating the gaps that Redis sought out to address in its implementation and design.

1. **Hybrid Consistency Models**: While many systems implement either eventual or strong consistency, there is a lack of efficient mechanisms to support both models simultaneously.

2. **Enhanced Fault Tolerance**: Existing solutions often struggle with efficiently promoting a new replica master upon node failure, which can lead to delays in game state updates.
3. **Optimized Replication Techniques**: Many distributed key-value stores face challenges in optimizing replication strategies, which can lead to increased latency in game state synchronization across players.
4. **Advanced Partitioning Strategies**: While partitioning is essential for scalability, many current systems do not efficiently manage data distribution during peak usage in online gaming.

1.2. Problem Statement

Our project aims to simulate a distributed key-value store that replicates the Redis cluster system, focusing on incorporating advanced features such as eventual and strong consistency, fault tolerance, replication, and partitioning. By creating this simulation, the project seeks to explore how these features operate within a distributed environment, providing a deeper understanding of their functionalities.

The significance of this simulation lies in the increasing demand for effective data management solutions, particularly in applications that require real-time data access and high performance, such as online gaming which is the real-world use case we have chosen. In these scenarios, ensuring data consistency and availability is crucial for providing a seamless user experience. While the project does not aim to address existing gaps in current systems, it serves as an important exercise in understanding the inner workings of a widely used distributed key-value store like Redis.

Through the simulation, the project indirectly addresses several problems that Redis aims to solve:

a. High Availability: Redis is designed to provide high availability through replication and automatic failover mechanisms. Simulating these features will enhance understanding of how to maintain data accessibility even in the face of failures.
b. Data Consistency: By exploring both eventual and strong consistency models, the project highlights the importance of maintaining data accuracy across distributed nodes, which is vital for applications where the integrity of game state is critical.
c. Scalability Challenges: Redis tackles scalability issues by allowing seamless horizontal scaling through partitioning. The simulation will examine how effective partitioning strategies can improve performance and manage growing data loads, particularly in resource-intensive applications like online gaming.
d. Latency Reduction: Redis is optimized for low-latency operations, making it suitable for real-time applications. The simulation will explore techniques to minimize latency in data access and updates, contributing to an enhanced user experience.

The potential implications of successfully simulating and evaluating a Redis distributed key-value store are the following:

a. Enhanced Understanding: By replicating the features of Redis, the project aims to deepen knowledge about how distributed key-value stores manage data consistency, availability, and fault tolerance.
b. Testing and Benchmarking: The simulation will allow for controlled testing and benchmarking of various configurations and features of the Redis system. This will provide valuable data on performance metrics such as throughput, latency, and resource utilization, which can inform best practices for real-world implementations.
c. Real-World Scenario Simulation: By simulating a Redis cluster, the project can recreate real-world scenarios commonly encountered in online gaming applications. This will help identify potential pitfalls and optimization strategies before deployment.

1.3. Objectives

The main objectives of our project are the following:

a. To design and implement a distributed key-value store that simulates the Redis cluster system.
b. To incorporate advanced features such as:
    i. Eventual consistency to ensure data availability while allowing for some latency in synchronization.
    ii. Strong consistency mechanisms to maintain data accuracy across distributed nodes in critical scenarios.
    iii. Fault tolerance capabilities, including the automatic promotion of a new replica master when the old master fails, ensuring system reliability.
c. To implement effective partitioning techniques that balance data distribution across nodes, improving scalability and performance during peak usage.
d. To evaluate the system's performance under various conditions, including different consistency models and failure scenarios, to assess its reliability and efficiency in real-time applications.
e. To explore techniques for data consistency, availability, and scalability in the context of online gaming applications
f. To document the findings and methodologies used throughout the project

2. Project Description

2.1 System Design

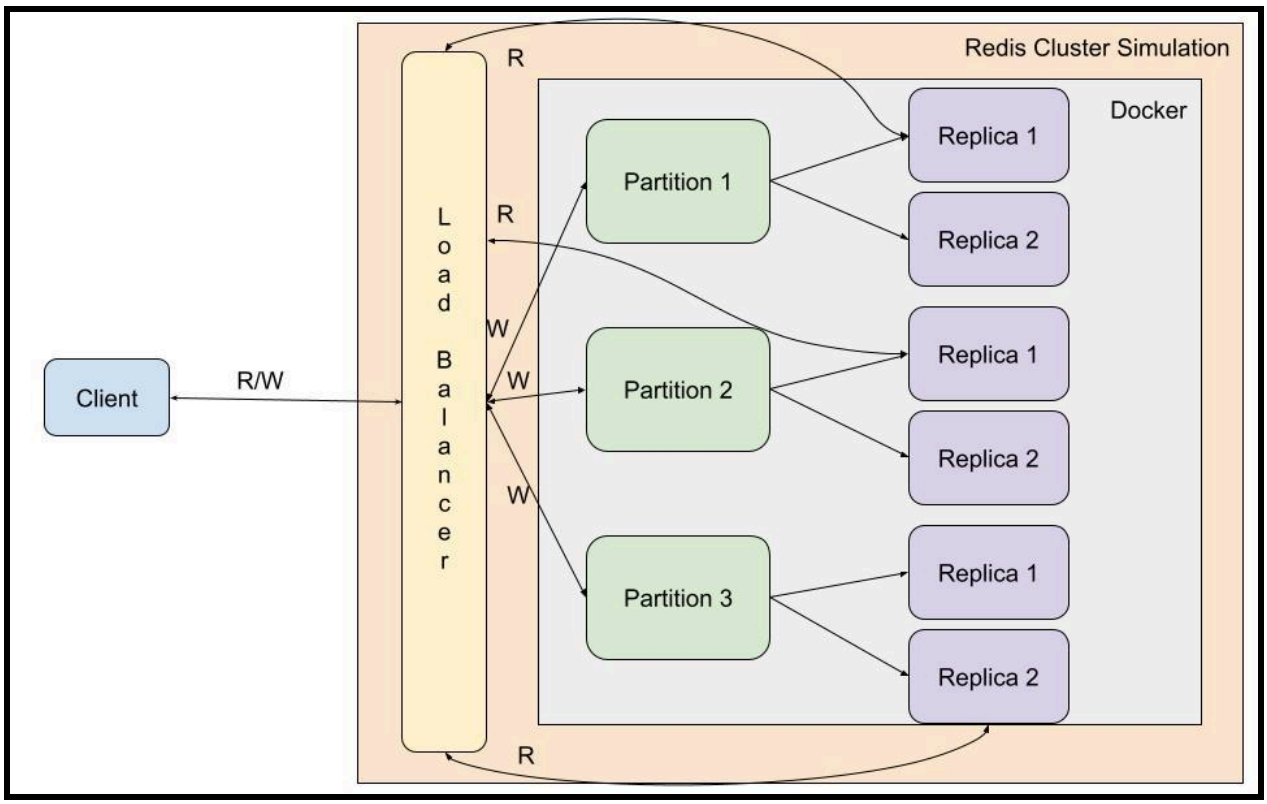

Fig 1. System Architecture Overview

A Redis cluster expands upon a single node Redis instance by adding in features that enable it to scale horizontally and operate in a distributed fashion. Redis Cluster adds partitioning, replication, fault tolerance and leverages the Raft consensus algorithm for fault tolerance and cluster consensus. Our project aims to simulate partitioning, replication and fault tolerance offered by the Redis Cluster.

As observed in Fig. 1, our project splits the single-node cache data into multiple partitions. A simple partitioning scheme based on the starting alphabet of the key could be used to partition the data.

To enable fault tolerance, each partition has multiple replicas. Redis clusters use single master replication, which is what this project simulates as well. As a result, requests that read values from a partition can be processed by replicas, but all write requests must be processed by the master partition.

Redis cluster uses eventual consistency to keep its replicas up to date. However this project also allows the cluster to support strong consistency.

The Raft algorithm in the Redis cluster ensures fault tolerance by monitoring the partition leaders and automatically electing a new leader when it detects issues. This project aims to simulate the same functionality without the Raft algorithm. Instead this simulation lazily checks the status of a current partition leader when it receives a request. If it doesn't receive a response from the leader, it simply routes the request to another active replica and notes it as the new partition leader. Hence the cluster's fault tolerance is built into the load balancer layer.

2.2 Implementation Plan

In a real distributed environment, partitions and replicas are placed in different machines (Ideally distributed in geographically different data centers). However, this project simplifies this deployment and simulates each partition and replica with a different Docker container. All Docker containers run on a single physical machine. Docker suits this purpose well as all containers run independently of each other, which will enable realistic network latency and faults.

All communication in this simulation (Client to Load Balancer, Load Balancer to replicas) will be done using HTTP. The project aims to use Python and the Flask web framework to have a web server listening and responding to HTTP requests.

The data stored in the replicas are stored in native Python dictionaries as key-value pairs. Python dictionaries suffice as they are in memory and there is no requirement to persist data onto secondary storage (Similar to how Redis stores data in-memory).

2.3 Data Strategy

This project aims to simulate a Redis cache for an online gaming use case. In particular this cache will serve the score (Value) for a queried gamerHandle (Key). This will be stored in a String to Integer python dictionary within the replicas.

The project does not use any data fetched from online repositories, rather it generates records and inserts them into the cluster.

3. Methodology

The project simulates the following advanced techniques from distributed systems,

3.1. Replication

The project simulates Redis Cluster's fault tolerance by having single master replicas. Slave replicas can respond to read requests, while write requests are serviced exclusively by the master replica.

3.2 Consistency

Redis Cluster offers only eventual consistency, however this project simulates both eventual and strong consistency. Strong consistency can be simulated by forcing the master to make update requests to every replica before responding the incoming write request, while eventual does not do so. Under eventual consistency the replicas are updated asynchronously and do not block the incoming write request.

The project plans to simulate this by using a text file as an event stream. On writes, the master replica appends an update entry into this stream and slave replicas poll this stream in fixed intervals to update their data stores.

3.3 Partitioning

This project supports partitioning by splitting the keyspace into multiple partitions. Various partitioning schemes could be used, for instance based on the starting letter of the key or the hash of the entire key. This project aims to use various partitioning schemes (Some that may deliberately introduce hot partitions) and measure their impact on query performance.

3.4 Fault Tolerance

The actual implementation of the Redis Cluster uses the Raft algorithm for fault tolerance. Nodes constantly check whether partition leaders are up and initiate an election for any leader that is unresponsive. This ensures that all partitions are accessible even if there are node failures, which ensures fault tolerance.

This project simulates fault tolerance by adding health checks into the load balancer instead. Rather than actively checking if leaders are responsive, the load balancer performs lazy checks whenever it needs to route a request to a specific partition's leader. If it detects unresponsiveness, it marks the current leader as down and notes another slave replica as the new leader. As a result the partition is still accessible despite node failures and fault tolerance is achieved.

4. Evaluation Plan

An evaluation plan acts as a feedback mechanism to improve the performance of the distributed system and to demonstrate the performance in numbers.

4.1 Metrics for Evaluation

- *Query execution time.*
    - Provides a glimpse into the implementation efficiency of the system.
- *Data spread ratio.*

    - An even data spread leads to better performance of the system overall.
- *Query execution time to # of concurrent users ratio*
    - Provides an idea of how well the system balances the load.
- *Network bottlenecks - query response transfer time.*
    - External factors affecting the performance of the system.

4.2 Expected Outcomes

Load balancer system demonstrating the following capabilities as part the implemented Redis system:

- *Fault tolerant mechanism.*
- *Eventual and strong consistency mechanism*
- *Data replication mechanism.*
- *Data partition mechanism.*
- *Concurrency handling mechanism*

5. Timeline

Milestone	Start Date	End Date	Team Member Responsible
Client-side development and Load balancer	10/13	10/20	Deebthik,Deva Dharshini
Core system development	10/20	11/10	SantoshGokul, Vikram
Evaluations and Reporting	11/24	12/02	Deebthik,Deva Dharshini,Santosh Gokul, Vikram

6. Conclusion

This project simulates a distributed key-value store modeled after the Redis cluster system, incorporating features such as partitioning, replication, fault tolerance, and support for both eventual and strong consistency models. Through this simulation, the project explores the behavior of distributed systems in real-world scenarios, particularly in online gaming applications, where high availability, data consistency, and low-latency performance are crucial.

The significance of this project lies in its contribution to understanding the internal workings of distributed key-value stores like Redis. By simulating core functionalities such as automatic failover, load balancing, and different consistency models, the project offers a practical exploration of scalability and fault tolerance challenges in distributed databases.

The potential impact of this project on the field of distributed database systems is considerable. It provides valuable insights into optimizing data replication, improving partitioning strategies, and managing consistency trade-offs in environments with high transaction throughput. Furthermore, the findings from the simulation can guide future research, enabling developers to design more efficient and resilient distributed systems that meet the growing demands of modern applications, particularly those requiring real-time data access and high performance.

7. References

1. E. A. Brewer, "CAP twelve years later: How the 'rules' have changed," IEEE Computer, vol. 45, no. 2, pp. 23-29, Feb. 2012.
2. A. Onişor, Redis in Action, Manning Publications, 2020.
3. A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44, no. 2, pp. 35-40, Apr. 2010.
4. D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in USENIX Annual Technical Conference, 2014.