# Ring Distributed Database (RingSync)

*for*

Cloud Computing (CS G527)

*by*

Kaustab Choudhury, 2020A7PS0013P
Shashank Shreedhar Bhatt, 2020A7PS0078P
Harsh Priyadarshi, 2020A7PS0110P

GitHub Repo: RingSync

## 1. Introduction

In the ever-evolving landscape of data management and storage, the advent of cloud-based object storage systems has redefined the paradigms of scalability, accessibility, and resilience. The essence of this project lies in the realisation of a robust and efficient distributed cloud-based object storage system that adheres to the principles of consistent hashing, replication, and distributed coordination. By leveraging a ring topology encompassing a vast $2^{64}$ ID space, the system is architected to partition and manage data across a network of nodes, where each node (running on a container) represents 500 virtual nodes.

The core objective is to ensure the seamless storage and retrieval of objects while maintaining high availability, fault tolerance, and data consistency. Replication strategies, encapsulating vector clocks, read-write quorums, and proactive measures such as read-repair and anti-entropy processes have been integrated to fortify the system against inconsistencies and data divergence, drawing inspiration from the foundational principles outlined in the Dynamo paper.

The system's functionality extends beyond mere data storage, encompassing a user-friendly REST API that empowers users to effortlessly create and delete objects. Furthermore, the API's design encapsulates the delivery of comprehensive JSON responses, including indicators of operation success or failure, vector clocks pertaining to all replicas, and crucially, the identification of the node responsible for the write operation.

This document delineates the design, architecture, and operational nuances of the proposed cloud-based object storage system. It outlines the interplay of components, strategies, and protocols aimed at realising the resilient, scalable, and performant solution to the given problem.

## 2. System Overview

### 2.1. System Architecture

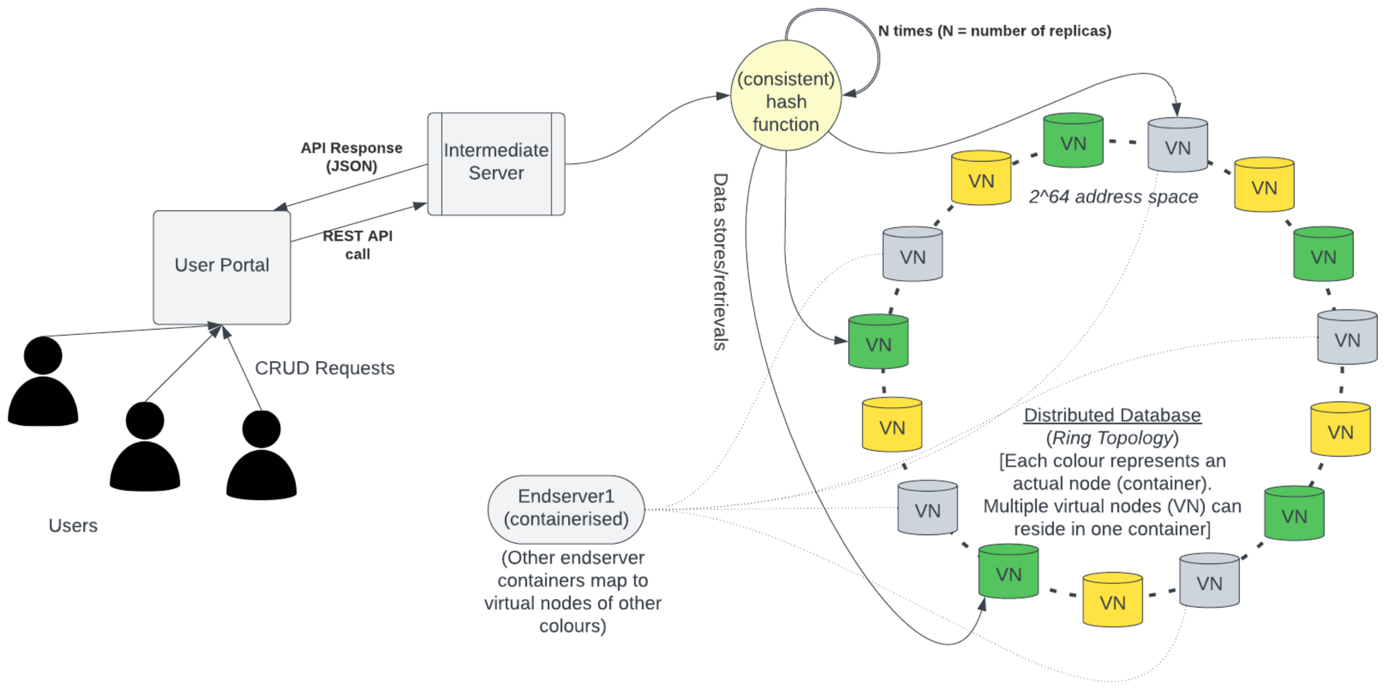
N times (N = number of replicas)
(consistent) hash function
API Response (JSON)
Intermediate Server
REST API call
User Portal
CRUD Requests
Users
Data stores/retrievals
VN
2^64 address space
Distributed Database
(Ring Topology)
[Each colour represents an actual node (container). Multiple virtual nodes (VN) can reside in one container]
Endserver1 (containerised)
(Other endserver containers map to virtual nodes of other colours)

Fig 1: Diagram showing how users' requests flow through the system's architecture

### 2.2. Key Components

- User portal
  - Client side application that enables the users to perform their desired CRUD operations on the distributed database.
  - Prompts the user to enter an option and performs the appropriate API call to the intermediate server accordingly by sending the user-entered information to that end.
  - Waits for the API call to execute and send back a reply. The reply is properly formatted and displayed to the user, as well as being stored for permanence as a .json file.
- Intermediate server
  - The code initialises a Flask server, defining various endpoints (/, /delete, /read, /create, /update) for the CRUD functionalities.

    - The code includes hashing functionality. Hash64(key) computes a list of hashes for a given key using MurmurHash3 library. It generates a series of hashes based on the specified parameters (N and ceiling).
    - Several routes (/delete, /read, /create, /register) perform actions related to interacting with servers based on the hashed keys and desired action coming from the user via the portal.
    - Server Registration and Ring Management: AddServerToRing(server) function adds servers to a consistent hashing ring, splitting each server into multiple virtual nodes and managing their placements in the ring structure. This is the virtual co-location of the ring topology. Linked list operations done on this ring using defined methods to merge and insert nodes into a linked list based on hash values and servers help in said management.
    - Server Initialization is performed including necessary variables, such as virtualNodeCount, N, ceiling, stepSize, serverList, and serverCount before starting the Flask application on port 6000 for intermediate server functionality.
- Endserver Initialiser

  The end initialiser's purpose is to launch a specified number of nodes (containers) based on the ring topology and register those nodes with the Intermediate server.
    - The *GetIP* method fetches the local IP address of the machine running the code by creating a socket connection to a remote host.
    - The *RemoveContainers* method empties the list of docker containers running on the machine.
    - The *AddContainers* method is the most important method here. It dynamically initialises the specified number of Docker containers as separate instances (endServer1, endServer2, etc.), assigning unique ports and registering each container's IP and port with an intermediate server for communication.

- Endserver

  This is the server at the end of the pipeline running the actual service of managing the data in our distributed database. One endserver running represents one actual node running on a Docker container, which can hold upto 500 virtual nodes within it.
  - Maintains a gRPC channel of communication with the intermediate server from which it gets the requests.
  - Services the requests based on whether it is create, read, delete, or update.
  - Sends back job status information such as vector clocks (for consistency) as well as data-related information, which is then forwarded to the user portal from the intermediate.

## 3. Design Rationale

### 3.1. Design Principles

- Separation of Concerns: Each code segment focuses on specific tasks, such as client-side interaction, server-side gRPC implementation, or handling CRUD operations. This ensures a clear separation of responsibilities and promotes modularity. It also falls in line with microservices architecture.
- Abstraction and Encapsulation: The code segments utilise functions, classes, and methods to encapsulate logic and functionalities. For instance, the server-side implementation abstracts the key-value storage operations, promoting reusability and maintainability. This is also in line with microservices architecture.
- Consistency: Threading and locking mechanisms are employed in the server implementation to handle concurrent access to the shared distributed database. A parallel update protocol has been implemented which ensures eventual consistency among the data replicas. Since multithreading in Python runs on a single thread (and the OS acts as a scheduler), locking mechanisms are employed in the server implementation to handle concurrent access to the shared distributed database. We make use of vector clocks to verify the consistency of the parallel operations across different replicas.

- Logging and Debugging: Logging mechanisms are integrated into the code segments to record events, errors, and debug information. This facilitates troubleshooting, monitoring, and understanding the code's execution flow.
- Modularity and Scalability: The code demonstrates modularity by breaking functionalities into reusable functions, classes, or methods. Additionally, the gRPC-based server design allows for scalability by enabling communication between distributed systems and services. This is following the principle of microservices architecture.
- Error Handling: Error handling mechanisms, such as checking response status codes or handling exceptions (e.g., socket errors), are implemented, ensuring robustness against potential failures and unexpected scenarios.
- Interface Design: The client-side application offers a user-friendly Command Line Interface (CLI) to interact with the server, simplifying user interactions and providing a straightforward way to perform CRUD operations.
- File-Based Data Persistence: The design employs a simple file-based (CSV) persistence mechanism for storing key-value data, offering a straightforward and accessible way to manage data without requiring a dedicated database management system.

### 3.2. Technology stack

- Docker
- Python
- Flask
- gRPC

## 4. Execution Instructions

4.1. You must have Docker and Python installed on your machine. We used Docker version 4.24 and Python version 3.10 for testing.

4.2. Make sure you have the following Python libraries installed:

```
- gRPCio
- gRPCio tools
- docker
- mmh3
- Flask
```

4.3. Clone the repository (either from the GitHub repo or by simply downloading the submitted folder).

4.4. Navigate to the directory of the repository. Create a Docker image named 'ringserver' via the following instruction:
`docker build -t ringserver .`

4.5. Run the intermediate server with the following instruction
`python intermediate.py`

4.6. Run the endserver initialiser script to register all N nodes (containers) with the intermediate server
`python endInitialiser.py`

4.7. You should have N containers running the *ringserver* image now. Now, any client can perform the desired CRUD operations by accessing the portal
`python portal.py`

Note: It might be needed to modify the IP addresses according to the necessities of the machines running this system on your end. Please do so accordingly before testing.

## 5. Conclusion

The implementation of a cloud-based object storage system represents a comprehensive solution built on key principles of distributed computing and efficient data management across a distributed storage. Leveraging the principles of consistent hashing and a ring topology with a vast 264 ID space, this system excels in scalability and fault tolerance. By distributing data across nodes—each node (Docker container) representing multiple (up to 500) virtual nodes—the architecture ensures both load balancing and resilience in the face of node failures. With configurable replication strategies and guaranteed consistency among replicas, this system fortifies data integrity while facilitating high availability.

The RESTful API serves as a user-friendly interface for clients, empowering users to create and delete objects seamlessly. It goes beyond mere functionality, offering transparent insights into operation outcomes. The API's JSON responses provide crucial details, including success or failure indicators, vector clocks for all replicas for a given data key, and the node number where the operations occurred. This granular information not only enhances user experience but also facilitates robust system monitoring and debugging.

Overall, this project's approach embodies the complex design principles learnt in the course, addressing key requirements while emphasising scalability, reliability, and transparency in cloud-based object storage.