Distributed Key-Value Store Using Conflict-Free Replicated Datatypes

Amitkumar Patel
New York University
ap7986@nyu.edu

Manas Vegi
New York University
mv2478@nyu.edu

Vashu Raghav
New York University
vr2326@nyu.edu

The paper presents a distributed key-value store using Conflict-free Replicated Data Types (CRDTs). We aim to create a reliable, scalable system for storing data in a distributed setup while ensuring high availability and eventual consistency. By leveraging CRDTs, we simplify concurrency control, enabling updates to occur in any order across distributed nodes without sacrificing consistency. Inspired by successful systems like Redis Cache, we employ CRDTs to achieve conflict-free writes and bidirectional replication among nodes. Our architecture utilizes a Map-based CRDT, with server replicas and clients maintaining local stores. Synchronization between clients and replicas is facilitated by a load balancer and REST API service utilizing ORSet and LWWRegister for conflict-free updates. Experiments show the system's effectiveness in maintaining consistency despite concurrent updates and replica failures.

Keywords: Distributed Key-Value Store, Conflict-free Replicated Data Types (CRDTs), Fault Tolerance, Eventual Consistency, Scalability.

1. INTRODUCTION

The main goal of our project is to design and implement a distributed key-value store utilizing Conflict-free Replicated Data Types (CRDTs). We aim to provide a robust, scalable, and fault-tolerant solution for storing and accessing key-value data in a distributed environment. By leveraging CRDTs, the goal is to achieve high availability and eventual consistency while allowing concurrent operations across distributed nodes.

One of the primary goals of distributed systems is to achieve consensus. Consensus, however, often requires a heavily involved protocol and/or significant amount of communication between the machines, which make it hard to implement and slow down the system respectively. With the relaxed constraints of eventual consistency, we can often trade the possibility of momentary divergence with much better performance and scalability. The divergence occurs when different operations are being applied at different machines concurrently.

CRDTs are one such eventual consistency solution that capitalize on achieving conflict resolution by making sure that the application of any operations is commutative. This is achieved because the algorithms have a deterministic way to resolve conflicts without having to coordinate with the other replicas, regardless of the order of the operations. The fact that we do not need to communicate to resolve conflicts allows the implementation to be simple while ensuring low latency. Not only is the implementation simpler, but we do not need a complicated fail-over protocol.

Implementing a key-value store using CRDTs was an interesting use case to try and implement as it's done at a gigantic scale by Redis to achieve sub-millisecond latency consensus [2]. To implement a Map as a CRDT, we need to utilize two underlying CRDTs as we will explain below.

2. LITERATURE SURVEY

To implement our proposed key-value store, we have done an extensive research literature review. The introduction of commutative replicated data types in "Consistency without concurrency control in large, dynamic systems"[1] highlights the commutative nature of CRDTs simplifies consistency maintenance, as it removes the need for complex concurrency control, allowing updates to execute in arbitrary orders while guaranteeing that replicas converge to the same result. The paper also mentions how two approaches in distributed systems dominate in practice. One ensures scalability by giving up consistency guarantees such as the Last-Writer-Wins (LWW) approach. The other guarantees consistency by serializing all updates, centralizing at a single database, or using state machine replication), which does not scale beyond a small cluster.

The paper "Approaches to Conflict-free Replicated Data Types" [3] mentions distinct approaches to designing CRDTs based on propagating operations or on the propagating state among the replicas and their benefits and tradeoffs. Operation-based CRDTs focus on sending individual actions to sync up replicas, while state-based CRDTs send the entire data set. Understanding these differences helps tailor CRDT design to suit different needs and situations.

The Redis cache, a geo-distributed low latency key-value store powered by CRDTs [2] enables the distributed system to do bidirectional replication among cluster nodes for conflicting concurrent write operations. These commutative nature and conflict-free write enable the Redis Cache to scale tremendously with high performance and low latency.

Inspired by a similar idea, we tried to develop our conflict-free distributed key-value store in achieving eventual consistency and fault tolerance in distributed systems. By building upon the insights and techniques developed in these systems, the proposed project aims to design and implement our own distributed key-value store that leverages CRDTs powered by our custom-designed data type for robust and scalable data replication.

3. PROPOSED IDEA

Our proposed architecture employs a Map as the wrapper key-value store CRDT. In our system, both server replicas and clients maintain their own local key-value stores in memory in the CRDT form. This in-memory storage approach enables key-value operations locally, including get, put, remove, and contains, without constant reliance on network connectivity.

To ensure consistency across the distributed system, clients periodically synchronize or merge their local stores with a remote replica's store. When initiating synchronization, a client sends a merge request to a server replica, which broadcasts the request to all replicas. The server replica waits for responses from a predefined quorum (f+1) before updating its store and responding to the client. This ensures that clients remain in sync with at least one remote server replica.

Our architecture is designed for scalability and adaptability. While our described setup involves maintaining key-value stores at both server replicas and clients, an alternative configuration can be implemented where clients are stateless, and key-value stores are solely managed by server replicas. This streamlined approach allows clients to directly send operations to any server replica, eliminating the need for intermediate synchronization steps and enhancing system efficiency.

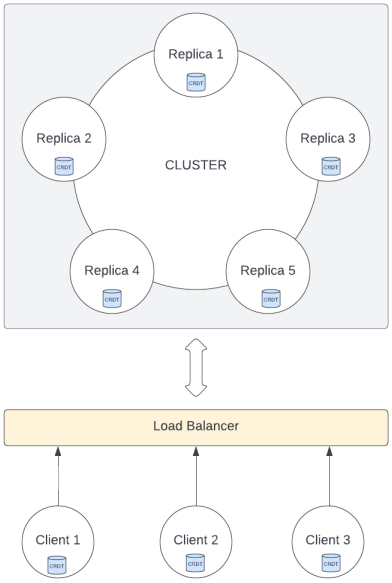

Fig. S1. System Architecture

4. IMPLEMENTATION

The system was implemented using the C++ programming language and the CRDT framework provided by the Github repository [5]. For the REST API service, we relied on Microsoft's cpprestsdk [6], and adapted a sample REST implementation from the GitHub repository [7]. These tools provided a robust foundation for developing our distributed system. The source code for our implementation is available on the Github repository CRDTs-Key-Value-Store and is open for community contributions. In the following subsections, we describe the implementation details, key considerations and challenges.

A. Underlying Datatypes

In our distributed system architecture, the ORSet [4] (Observed Remove Set) and LWWRegister (Last-Write-Wins Register) are integral components of the Map data structure, collectively ensuring conflict-free updates and consistent value storage across distributed replicas. The ORSet facilitates concurrent addition and removal of keys without conflicts, leveraging timestamp-based ordering for conflict resolution. On the other hand, the LWWRegister enforces the "last-write-wins" policy for values associated with keys, ensuring deterministic conflict resolution and consistent value storage. Integrated into the Map, these CRDT-based data structures enable efficient key management and value storage, with functions such as adding, removing, and retrieving key-value pairs while preserving consistency and convergence across distributed replicas.

B. Server Replicas

Our server replicas are at the core of our distributed system, handling client requests, processing data, and maintaining consistency across nodes. When a request to update the shared state, like a merge operation, is received, the server broadcasts it to all other replicas. It waits for (f+1 responses), where f is the maximum number of potential faulty replicas, before proceeding. This ensures that updates are distributed uniformly and that the system maintains integrity, even in the face of faults or network issues.

Implementing efficient asynchronous task handling in C++ presented a unique challenge due to the language's lower-level nature and lack of built-in support for asynchronous programming. We overcame this challenge by leveraging futures to manage concurrent tasks, allowing us to broadcast client requests to all replicas concurrently while waiting for (f+1) successful responses.

C. Clients

On the client side, our implementation handles interactions with server replicas, including read and write operations, data querying, and response processing. Clients send requests to the load balancer, which forwards them to the appropriate server replica based on predefined routing logic. These requests may include operations such as putting, getting, or removing key-value pairs from the distributed map. Upon receiving responses from server replicas, clients extract relevant information and handle any errors or exceptions, including data queried from the distributed map or acknowledgment of successful operations.

D. Load Balancer

The load balancer serves as a centralized entry point for client requests and facilitates efficient workload distribution across server replicas. It employs custom routing logic to distribute incoming requests, dynamically adjusting routing decisions based on server availability. For each sync operation, the client connects to the Load Balancer to request a replica to talk to. Once that is done, the client directly connects to the replica to communicate.

E. Server Replica Fail-Recover Simulation

We implemented a bash script to simulate failures in up to two server replicas at any given instant, mimicking real-world scenarios of node unavailability. Upon failure, the server replicas were not restored to their previous state but reinitialized. Server replica state recovery naturally occurs when the node receives broadcast messages from other replicas or merge requests initiated by clients.

F. Client Command Generation

To automate the behavior of the client, we programmed a bash script for the creation of commands for a client. Given a client ID and the desired number of commands, it generates a sequence of random commands, such as adding, removing, and retrieving key-value pairs, as well as listing all pairs and pausing execution. The script then directs these commands to the main program for execution, facilitating automated testing and evaluation of the distributed system's functionality.

All communication within the system is facilitated through REST API calls across server, client, and load balancer instances running on the same machine, providing a comprehensive environment for development and testing. Looking ahead, deploying this system in a distributed setup in the cloud would be desirable for testing the performance of the distributed key-value store. This would allow testing our system in the presence of real network delays and the distributed nature of operations.

5. RESULTS AND ANALYSIS

The experiments were done on a local machine with 5 different processes running on different ports to mimic distributed replicas. There was also one process for the

LoadBalancer and 1-3 client processes. For each test, bash scripts were run to automate the thousands of client operations necessary to populate the client CRDT Map before calling the sync operation. One of the scripts running also killed and recovered upto 2 replicas to model a fail-recover setting.

To test for correctness, we made a call to the cluster which would get the states (map) of each replica and compare it for equality. The way we checked for equality was a simple diff on the serialization of the Map. At the end of every experiment, we did this diff check on the serialized maps of each replica. From sizes 1k, 2k, 10k, 50k, and 100k, the replicas were all consistent at the end of the experiment. This we believe is a good result showing that the protocol is indeed eventually consistent.

Another point to note is that there was no additional fail-over protocol specified. The moment the replica comes back up and initializes an empty CRDT map, when it receives another CRDT map for syncing with its own, it is back to being synced. This makes implementation really simple and enables for a graceful recovery of replicas.

Running the experiments locally on an M1 MacBook Air had the following results: For a sync involving a key-value store of 1000 entries, each operation took around 200ms. For a sync involving of a key-value store of 100,000 entries, each operation took around 1.8s.

The time taken for the sync operations was larger than expected. This is most likely because we chose to apply state-based sync operations, i.e., the entire state (key-value store) is sent over the network to the replicas and all communication among the replicas also occurs by sending over the whole key-value store each time. Since the communication step scales with the size of the key-value store, so does the time taken by the sync operation.

Another possible reason for the higher-than-expected latency was because of a slow implementation of the serialization/deserialization of the Map CRDT. Moreover, the time taken for serialization/deserialization scales with the size of the data structure, thus explaining the rise in the latency with a larger key-value store.

On the positive side, the client operations that were taking place offline (without communication with the servers) were pretty much instant as a result of being local. In real-world implementations, syncing among the servers is done as a background process and we could implement the sync operation to involve the client replicating its CRDT on just one replica. This way, the client becomes consistent a bit later (the next time it talks to the server) but the communication step will only be 1 RTT, hence making the flow more responsive for the client.

One other possibility to consider is operation-based CRDTs, which we did not implement due to the time frame we had for the project.

6. CONCLUSION

Our project demonstrates the robustness of CRDTs in achieving eventual consistency within distributed key-value stores. Through rigorous experimentation and analysis, we validated the reliability of our protocol under various conditions, including concurrent updates and fail-recovery scenarios. Based on our results, state-based CRDTs excel with smaller state management, but for larger state, operation-based or delta-based CRDTs may be preferable to minimize communication and serialization overhead.

7. FUTURE DEVELOPMENTS

The future roadmap for the project is to evaluate an operation-based or delta-based CRDTs. The operation-based or delta-based CRDTs could have smaller messages being transmitted and quicker serialization with some other tradeoffs (Eg: delta-based might not work with fail-recover model). Further, we would explore more efficient serialization and deserialization techniques for the CRDT maps to reduce network latency.Before implementing these enhancements, we plan to evaluate our current approach on the cloud to assess its performance and scalability in a real-world distributed environment.

REFERENCES

1. Letia et al, Consistency without concurrency control in large, dynamic systems, *Source:* https://asc.di.fct.unl.pt/~nmp/pubs/osr-2010.pdf
2. Under the Hood: Redis CRDTs (Conflict-free Replicated Data Types), *Source:* https://redis.io/wp-content/uploads/2021/08/WP-Redis-Redis-Conflict-free-Replicated-Data-Types.pdf
3. PAULO SÉRGIO ALMEIDA, Approaches to Conflict-free Replicated Data Types, *Source:* https://arxiv.org/pdf/2310.18220
4. Bieniusa A, Zawirski M, Preguiça N, Shapiro M, Baquero C, Balegas V, Duarte S. (2012). - An Optimized Conflict-free Replicated Set. *Source:* https://arxiv.org/pdf/1210.3368
5. GitHub CRDT Implementation *Source:* https://github.com/miladghaznavi/crdts
6. Microsoft's C++ REST SDK *Source:* https://github.com/microsoft/cpprestsdk
7. GitHub C++ REST SDK Sample Client *Source:* https://github.com/Meenapintu/Restweb/tree/mac