White Paper: Designing a Fault-Tolerant and Scalable Data Replication System

Authors: Everest K C, Jake Forsberg

Architectural Priorities

To ensure the reliability and scalability of our data replication system, we prioritize the following architectural requirements in the following order: **Fault Tolerance**, **Elasticity, Scalability, Consistency, Availability, and Performance.**

Since, we plan to use strong consistency built into the Redis, we will be compromising some of availability, scalability and performance. When we use strong consistency no read operations can be performed on the cluster until the data has been written to all the replicas.

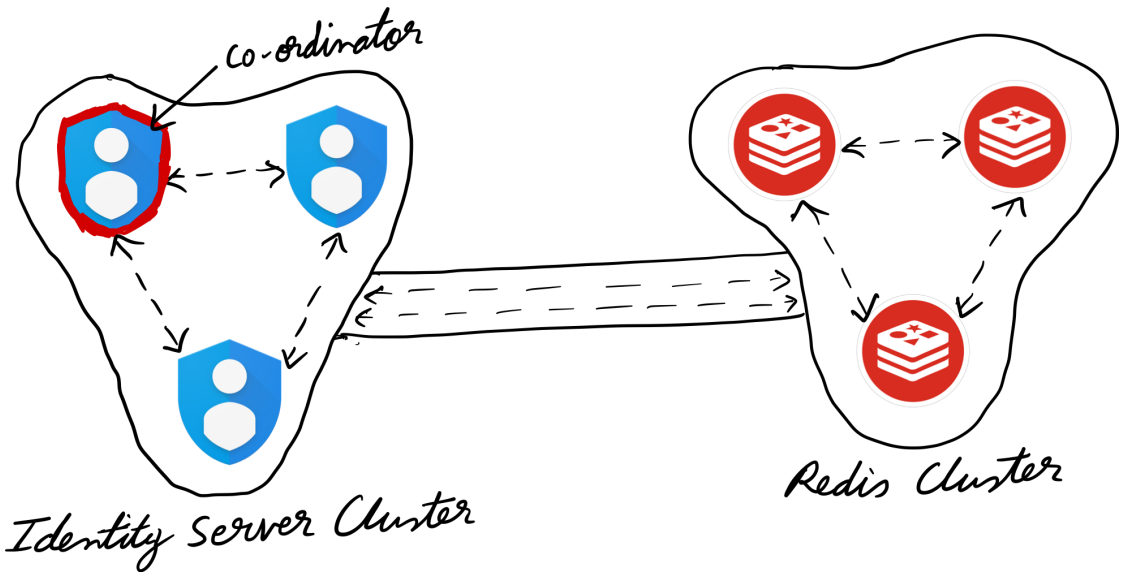

Figure 1: High Level Architecture Diagram of Fault-Tolerant and Scalable Identity Service

Our Identity Service architecture consists of two clusters - one for the Identity Server (Backend) and one for Redis. To take advantage of Redis' built-in distributed properties, we've separated the service into Backend and Data Service clusters. This allows us to manage the distributed properties of the Identity Server Cluster more effectively.

Fault Tolerance

We have designed our system to continue functioning even if one or more of its components fail. We will be using redundancy, failover mechanisms, and backups to ensure that our system can handle any unexpected failures.

- Redundancy: We use multiple redundant servers. Our approach involves deploying at least three servers to achieve fault tolerance from failure of one server (using 2n+1 rule). We supply the clients with a list of IP addresses of all the available servers in the cluster, so that if one connection fails the client can quickly make a request to another server in the list.

- Failover Mechanism: Our failover mechanism ensures that when a worker node fails, the coordinator immediately removes it from the system. Similarly, when the coordinator fails, an election is held to select a new coordinator. We treat both node failure and connection failure in the same manner.
Whenever a failed node comes back online and contacts any other node in the cluster, it is listed as an active node, and the workload is redistributed to it as well. If a failed coordinator comes back online, it will be demoted to a normal node and will start working like any other node in the cluster.
The Redis Cluster has its own built-in failover mechanism and we don't need to worry about it a lot.

- Backup: In terms of backup, we have taken measures to ensure that our Redis cluster is fault-tolerant. We have implemented the AOF (append-only file) persistence mechanism in Redis, which writes all data to a file that can be used to restore a failed node. This ensures that we can quickly and easily recover from node failures and maintain the integrity of our system.

Elasticity
Our system is designed to be capable of dynamically adding or removing servers from the cluster during runtime. When a server is added to the cluster, the coordinator will detect this and add it to its list of available servers and will route clients to it based on the round robin system. In a similar way, when a server is removed from the cluster the coordinator will remove the server from its list of available workers and will no longer send any requests towards it.

Consistency
We plan to have a strong consistency in this system. While all of our worker servers are allowed to conduct both reads and writes, we plan to use a combination of distributed locks and transactions to ensure that we cannot modify data at the same time, whether or not they are carried out on the same server. We will also combine the transactions with the redis WAIT command. This command blocks the redis client until the write commands are successfully transferred and acknowledged by the Redis replicas. However, as long as a write is not occurring, reads will be allowed at all times.

Scalability
With our consistency model, adding more nodes should give us an increase in performance as we could handle more reads concurrently in the system. However, providing more nodes and traffic will negatively impact our performance when we are performing write operations, as we are limited to one at a time.

Availability
Unless a server is carrying out a write on the redis cluster or our coordinator goes down, our service should be highly available. Reads are allowed concurrently any time a write is not being carried out, meaning we can service many reads at once helping our availability. However if a write occurs, then all new requests are halted until that write completes which will impact availability. If a coordinator goes down, then our service will not be available during the time an election is carried out until a new coordinator can be appointed.

Performance
Due to our strong consistency, we will sacrifice our performance on writes, as it locks any server from carrying out any form of requests on our Redis server. However reads without any writes should still be performed to a reasonable speed as these can be carried out in parallel. Performance also may be impacted

by our choice to have a coordinator act as a load balancer, forcing any client request to go through it before it is allowed to contact a worker server and perform its request.

Communication Model

For Identity Server Cluster

To improve the fault tolerance and consistency of our system, we've decided to use a coordinator in our communication model. The coordinator will act as a centralized entity responsible for managing the communication between the multiple servers.

In our design, the coordinator will be chosen based on the Bully Algorithm, which is an election process initiated when the current coordinator fails or becomes unavailable. The servers will communicate with each other and elect a new coordinator based on the highest process ID of all available servers. Once the new coordinator is elected, all servers will switch to communicate with the new coordinator.

The coordinator will play a crucial role in our system by managing the replication of addresses of all the servers in the cluster. Whenever a client attempts to contact the cluster, it will be forwarded to the coordinator. The coordinator will then return a list of available worker nodes to connect to in a round-robin style, acting as a load balancer for the server. The coordinator will also handle replicating and managing worker nodes.

In the event of server crashes or network communication failures, the coordinator will take charge of managing the fault. If a server becomes unavailable, the coordinator will identify the failed server and remove it from the system. The coordinator will then redistribute the workload to the remaining servers. If the coordinator itself fails, the election process will be initiated, and a new coordinator will be elected.

For Redis Cluster

Communication within Redis is slightly different as there is no coordinator. Each node in Redis acts as a combination of Master and Worker nodes, and the data is partitioned between the nodes. When one of our worker nodes communicates with the Redis cluster, Redisson (our Redis Client) uses hash shard slots to determine which Redis node to communicate with. This combination of Master/Worker exists for fault tolerance reasons.

(Extra Credit)

1. Load Balancing: The co-ordinator identity server maintains a list of addresses of all the available nodes in the cluster. When a clients' initial request is forwarded to the server, the server returns with the list of available nodes in a round-robin fashion. The client can then connect to the server based on the list it received.

2. Monitoring and Alerting: We plan to use Prometheus to collect information from all the nodes in our system and use Grafana to visualize the data into a nice informative Dashboard. We shall also use the alerting system from Grafana to send out emails when any of your nodes in the cluster fails.