Designing and Implementing Multi-user Distributed Text Editor

# Introduction

It is required to design and implement a multi-user distributed text editor, which will allow several users to collaborate in reading and editing text documents in a real-time environment where changes at any user will be broadcasted to all the other users at the same exact time of the changes. It is more like we need to design and implement a clone of Google Documents application.

# Project Description:

# Beneficiaries of the project:

# Roles of the team:

# Task Breakdown structure:

# System Architecture:

# System Design

There are several functional and non-functional requirements that we must follow in order to start our system design plan.

## Functional Requirements

* Users can change a document at the same time without any conflict
* Allow sharing documents between users through a unique document ID and hyper-link

## Non-Functional Requirements

* The system must support multiple clients or autonomous agents like an API for sharing and updating data.
* The system should be distributed across multiple clients or server nodes.
* The system should be robust
* The system should be able to continue operation even if one of the participant nodes crashes
  + If three clients are collaborating on the same document and one client failed, the other two should continue collaboration on the system achieving reliability.
* It should be possible to recover the state of the node following a crash, to continue operation
  + Retrieving the final state of the document when the node goes back online
* The system should maintain multiple replicas for fault tolerance.

## Design Constraints

* Concurrency
  + Since several users are working on the same document
  + Operational Transformation
* Latency
  + Clients are working in different places, and the connection is established through the internet, so there is a latency between each and all clients when they are collaborating on the same document.
  + RESTful vs Publish/Subscribe Architectures.
* Security
* For each type of user, what type of data access restrictions are required.
* For each type of user, what type of update privileges are required.
* For each type of user are there any other Window behaviors which require specific privilege.

* Backup and Recovery Requirements
* Acceptable down time for system.
* Acceptable data and user interface state loss due to system crash*.*

## Operational Transformation

In order to provide real-time and collaborative environment in a text editor, we must consider any conflicts that may arise when more than two nodes are collaborating. For example, if a node inserted some text at position x, and another node deleted the text that exists at position x at the same time. Here, we present Operational Transformation, which is a technology that aims to solve conflicts in real-time collaborative editing environments. In order to do that, we must maintain consistency between local replicas of documents, since each client have its own local copy of the document.

A document will be stored as a sequence of operations in order of execution instead of plain text. So, we need a collaboration protocol to understand when to apply changes. We thought about identifying possible operations into three types:

* Insert Text
* Delete Text

Graphical user interface, application

Description automatically generatedWhenever we edit a document, all the changes are appended to the document saving these operations in one of those three types. In addition to saving operations by each user in a changelog database.

Figure 1 - Operational Transformation Sequence diagram

## WebSocket vs HTTP

1. Web Socket

* WebSocket is a technology that enables bidirectional, full-duplex communication between client and server over a persistent, single-socket connection. This allows for low-latency, real-time updates, and the creation of richer communication and gaming applications. Previously, the web was dependent on requests and responses, which aren’t dynamic enough for those kinds of apps.
* WebSockets generally do not use XMLHttpRequest, and as such, headers are not sent every-time we need to get more information from the server. This, in turn, reduces the expensive data loads being sent to the server.
* WebSocket is an event-driven protocol, which means you can actually use it for truly real-time communication. Unlike HTTP, where you have to constantly request updates, with websocket, updates are sent immediately when they are available.

1. Web Transport

* The newer WebTransport offers secure, multiplexed, realtime transport and already has APIs for sending data both reliably and unreliably. In a reliable data transfer, the sender is notified of the success or failure of the data transmission, and failed transmissions are usually resent until they succeed, after which the next data packet is sent. In unreliable transfer, there’s no confirmation of transmission success, and packets that aren’t received simply don’t get delivered.
* Unreliable transfer is often used for things like streaming videos, where speed is a concern, and minor data loss, such as a few frames of video, is acceptable. Because WebTransport uses both of these methods, there are many use cases for it, such as bidirectional data streaming for multiplayer gaming, interactive live streams, and data transfer for sensors and internet of things devices.
* It avoids the head-of-line blocking delays that WebSocket suffers from, and is less resource intensive when creating connections
* It works with HTTP/3, the upcoming version of the transport protocol used by the World Wide Web. HTTP/3 uses the QUIC protocol for transport layer data exchange, which has several advantages. QUIC can prevent head-of-line blocking delays, improving network performance in many situations. This is a limitation of WebSocket.

# RESTful vs Event-Driven Architectures

# General architecture for collaborative editing applications

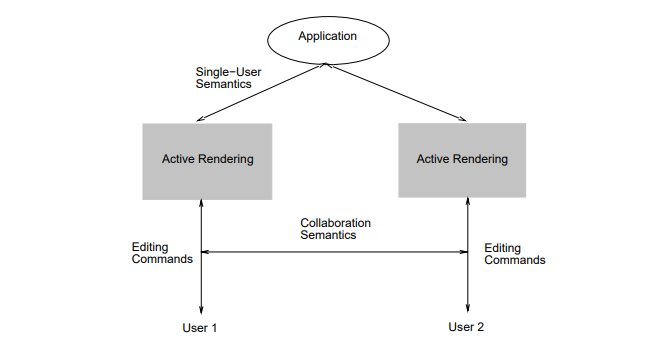


Figure 2 – general architecture for collaborative editing apps

# 1st: RESTful Architecture (representational state transfer)

## Characteristics

REST principles are defined by four interface controls, including identifying resources, managing resources through representations, self-descriptive communications, and hypermedia as the engine of the application state.

View the distributed system as a collection of resources, individually managed by components and these resources can be added, removed, retrieved or modified by remote applications while keeping that these resources provide the same interface and are identified by the same naming scheme.

The messages sent to or from a specific service are fully described and after executing an operation at a service that component totally forgets about the caller.

Has greater **stability** because it restrains component performance. so that each component can’t see further than the immediate layer with which it is intermingling.

REST uses less bandwidth, simple and more flexible making it more useful for internet usage.

Uses http operations (GET, POST, PUT, DELETE, UPDATE, PATCH)

## Guiding principles for REST (constraints)

1. Its **layered system** allows generating a more scalable and flexible application. An application has better security due to its layered system, as components in each layer can’t interact outside the successive layer. Also, it balances loads and offers shared caches for stimulating scalability.
2. **Code on demand (optional):**

A REST API definition permits extending client functionality by downloading and implementing coding in the form of applets or scripts. This restructures clients by decreasing the number of features important to be pre-implemented.

This REST principle allows for applets to be communicated through the API used within the application.

1. **Uniform Interface**

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions.

**Constraints for applying the principle of Uniform Interface:**

* Identification of resources
* **Manipulation of resources through representations 🡪** The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server.
* **Self-descriptive messages**
* **Hypermedia as the engine of application state🡪 The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.**

1. **Client-Server**

separation of concerns, By separating the user interface concerns (client) from the data storage concerns (server), we improve the **portability** of the user interface across multiple platforms and improve **scalability** by simplifying the server components.

1. **Stateless**

The server cannot take advantage of any previously stored context information on the server. For this reason, the client application must entirely keep the session state

1. **Cacheable**

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable. And If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

## How it works

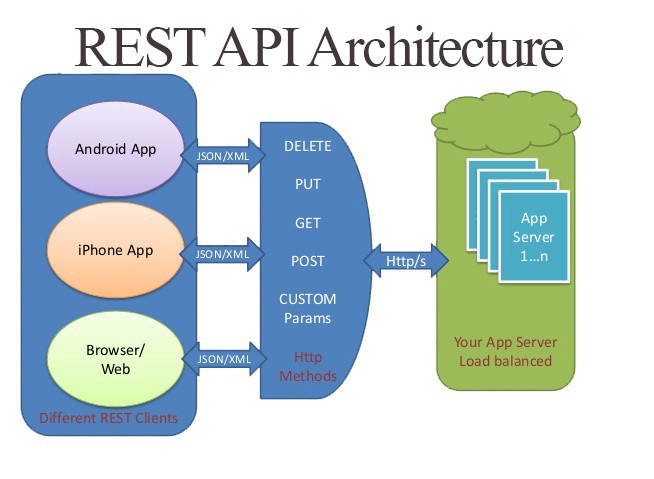


Figure 3 - REST architecture

A request is sent from client to server in the form of web URL as HTTP GET or POST or PUT or DELETE request. After that, a response comes back from server in the form of a resource which can be anything like HTML, XML, Image or JSON. But now JSON is the most popular format being used in Web Services.

## When to use REST?

* + we need a time-bound request/reply interface
  + Convenient support for transactions
  + our API is available to the public
  + the project is small (REST is much simpler to set up and deploy)

# 2nd: Event-Driven architecture

As most of the system design architects used REST architecture as their service communication layer, but further more and more Projects used Event-Driven Architecture due to its pros which we will discuss in this section.

## Event-Driven detailed architectural design.

Figure 4 - event-driven architecture

## Main Components:

Event-Driven *Software* Architecture, describes various logical components and their roles in events generation, transmission, processing, and consumption.

An event-driven architecture mainly consists of four components:

**1**. Event🡪The change in the state of an object that occurs when users take a specific action.

**2**. Event Handler🡪A software routine, which handles the occurrence of an event.

**3**. Event Loop🡪Handles the flow of interaction between an event and the event handler.

**4**. Event Flow Layers🡪The event flow layer is built on three logical layers: Event Producer, Event Consumer, and Event Channel (Event Bus).

Producer🡪Responsible for detecting and generating events.

Consumer🡪 Consumes the events produced by the event producer.

Event Channel🡪Transfers events from the event generator to the event consumer.

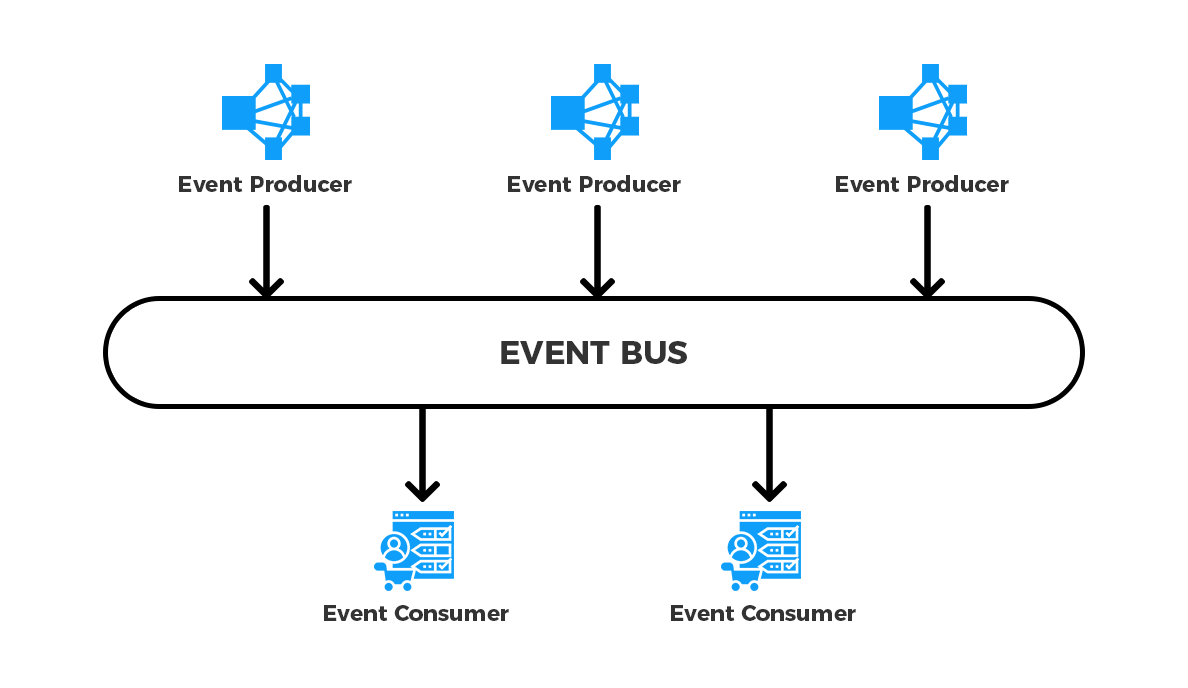


Figure 5 - event channel illustration

## Characteristics

Event-driven distributed systems have two essential characteristics which differentiate them from other system architecture types:

1. The existence of several software/hardware components that run simultaneously on different inter-networked nodes.
2. The use of events as the main vehicle to organize component intercommunication.

## Advantages:

* Adding new events and processes is very easy in the event-driven architecture.
* Event-driven architecture also gives the transactional guarantee, getting notified of every successful transaction that occurs.
* The architecture is easily replaceable.
* We can roll-back changes or move to any event in the event-driven architecture. Which is helpful in case any issue occurs.
* Highly responsive. Instead of waiting for issues to occur, you can easily detect them in advance therefore ensuring that the app keeps working.

## How It works?

We have to define what is an event first, an event is simply *put,* a significant change in state, which is triggered when a user or service takes an action. Other services consume such events in order to complete any tasks that arise as a result of that event.

**Unlike with REST**, services that create requests do not need to know the details of the services consuming the requests.

## Difference from REST architectures

An event-driven architecture offers several advantages over REST, Including:

* **Asynchronous**🡪 Event-based architectures are non-blocking and asynchronous. This permits resources to go on to the next task without worrying about what happened before or what will happen next once their unit of work is completed. They also allow events to be queued or buffered, preventing consumers from applying pressure or blocking to producers.
* **Loose** **coupling**🡪 *services operate independently,* without knowledge of other services, including their implementation details and transport protocol.
* **Easy Scaling**🡪 Due to the decoupling of services in an event-driven design, and the fact that services typically execute only one activity, tracking bottlenecks and scaling that service becomes simple.
* **Recovery Support**🡪 With the queue of Event-Driven Architecture, recovery of lost work by replaying events from the past. This can be Important to prevent data loss when a consumer needs to recover.

## Difference from Request-Response architectures

|  |  |  |
| --- | --- | --- |
|  | Event-Driven | Request-Driven |
| When action is taken? | According to context. | When request is being processed. |
| Why action been taken? | Triggered by situation being detected. | As a response to a specific request. |
| What happens when an event/request occurs? | Event can be ignored, increment the state, trigger an internal derived event or trigger a situation. | Response is always produced. |

## Issues To always Consider

Despite the event-driven architecture offers many benefits, it is very complex to implement and needs a lot of essential concepts to keep in mind when implementing this architecture.

1. **Increasing Complexity**

Since, Event-Driven Systems are *loosely coupled and Highly Distributed,* it’s difficult to tell which event is linked to which microservice and what the relationship between them is and we may never know when a small change can result in an unexpected chain of reactions.

1. **Not much control**

It’s not clearly defined which event should be consumed and which shouldn’t. there are some actions that we may not want to publish, such as a failures or bugs. The issue is how can a microservice know that transaction that we don’t want to publish, This adds an extra layer of confusion.

1. **Certain Events can’t be Undone**

While we can quickly undo modifications in Event-Driven Systems, some events are beyond our control. It's just not possible due to the event's reliance on a third-party system. we can't, for example, undo an already sent email.

## Event-Driven Implementation

## Load Balancer

Load balancing refers to the act of distributing network traffic across multiple services.

This makes sure that there's not too much load on a single server which could cause it to crash.

A load balancer acts as a ‘reverse-proxy’ to represent the application servers to the client through a virtual IP address (VIP). This technology is known as server load balancing (SLB). SLB is designed for pools of application servers within a single site or local area network (LAN).

Load balancers health check the application on the server to determine its availability. If the health check fails, the load balancer takes that instance of the application out of its pool of available servers. When the application comes back online, the health check validates its availability and the server is put back into the availability pool.

Because the load balancer is sitting in between the client and application server and managing the connection, it has the ability to perform other functions. The load balancer can perform content switching, provide content-based security like web application firewalls (WAF), and authentication enhancements like two factor authentication (2FA).

This is the primary function of the load balancer, server load balancing (SLB). The agent can provide additional functionality based on their role in the conversation. They can decide to allow and/or deny certain details (security). They may want to validate that the person they are talking to (authentication).

### Benefits of using Load Balancer

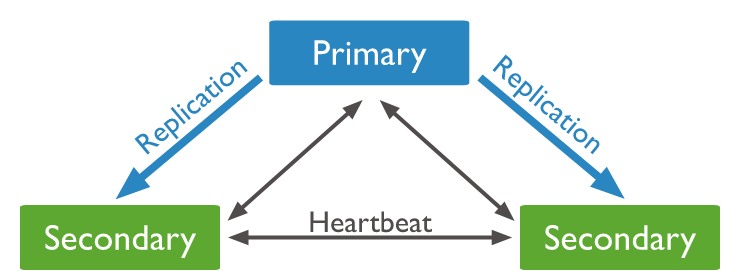
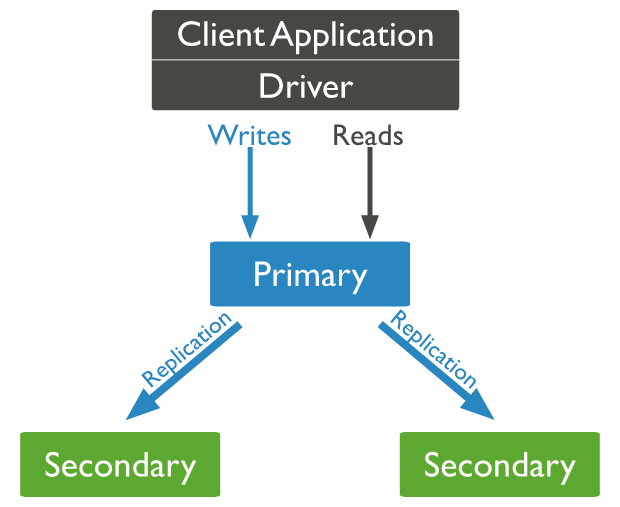
* it helps improve the responsiveness of your application.
* It also limits the chances of servers crashing as they're not being subjected to loads beyond what they're capable of withstanding.
* This is done to ensure maximum speed and capacity utilization

## Load balancer Implementation in Multi-Users Distributed Text Editor

# Database Replication

Due to the crucial component in the distributed systems applicationswhich is **Data Availability,** data base replication is the way to address this component.

**Database Replication in distributed systems applications** is the process of building multiple copies of data and store them in different locations for mainly the sake of **backup,** similarly to data mirroring, data replication can be applied to both individual computers and servers, it also refers to Data distribution from a source server to other servers while being updated and synced with the source so that users can access data relevant to their activities without interfering with the work of others.



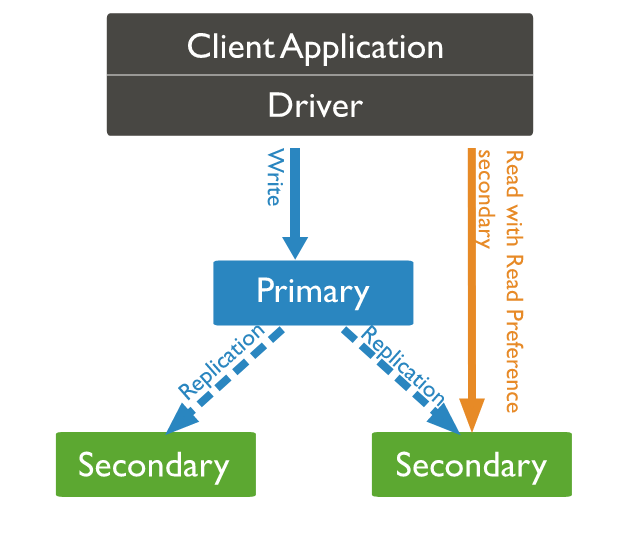
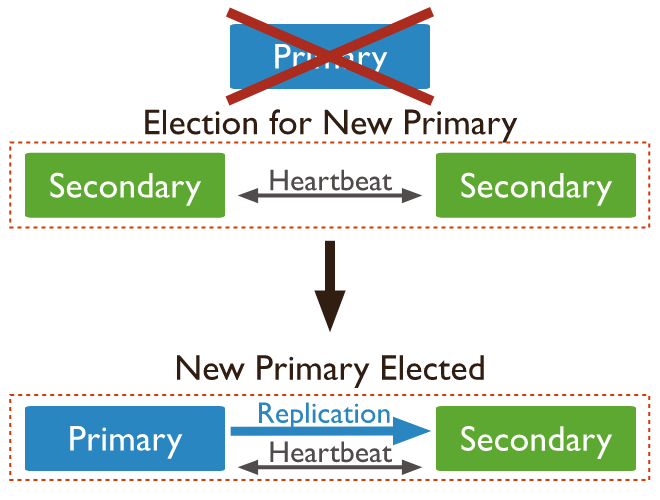


Figure 6 - database replication illustration

Data replicates can be stored:

* Within the same system
* Onsite and offsite hosts
* Cloud-based hosts

## Importance of Database replication in distributed systems:

1. **High availability** 🡪 as we have mentioned that the data availability is the crucial component for a successful distributed system application and it’s applied by implementing the concept of data replication as it is the most effective way to increase the data availability, data can be also replicated over multiple locations so that the user can be able to access it if some copies became unavailable or lost as a result of any site failures.
2. **Fault tolerance** 🡪 when any network/system/site fails occurs, the system succeeds to operate as when a replica fails, the service can be served by another replica.
3. **Read Scalability** 🡪 read queries can be serviced from copies of the same data that have been already replicated and this helps to boost the overall throughput of queries
4. **Reduce Latency** 🡪 that can be applied when keeping data **geographically** closer to the user, replication helps to reduce data query latency. Example is *CDNs (Content Delivery Networks)* whose applications (i.e. Netflix) succeed to retain a copy of duplicated data closer to the user.

## Common Types of Data Replication in distributed systems:

1. **Asynchronous vs Synchronous Replication**

|  |  |
| --- | --- |
| Asynchronous Replication | Synchronous Replication |
| The replica gets modified after the commit is done onto the database. | The replica gets modified immediately after some changes are made in the relation table. |

1. **Based on Server Model**

|  |  |  |
| --- | --- | --- |
| **Single Leader Architecture** | **Multi Leader Architecture** | No Leader Architecture |
| One server accepts client writes and replicas pull data from it. | Multiple servers can accept writes and serve as a model for replicas. | Every server can receive writes and work as a replica model. |
| It’s a synchronous technique but It’s quite rigid. | Leaders should be in close proximity to all of the copies to avoid delays. | Despite it provides maximum flexibility, it makes synchronization difficult |

## Advantages of data replication in distributed systems:

* + **Enhanced Performance:** Users can obtain data from the server nearest to them because the same data is stored in multiple locations, reducing network latency and increasing speed.
  + **Increased Availability:** Multiple users are allowed to manage and view data without them interfering with each other.
  + **Allows Multiple User Access:** for query-execution when multiple users accessing.
  + **Ensures business continuity:** when site/network failures in important business systems, crucial data is never lost but van be recovered and business in maintained to continue.

## Disadvantages of data replication in distributed systems:

* + **Large storage space:** especially when using full replication technique where many copies need to be synchronized and updated which may lead to high costs and reduced performance.
  + **Maintenance costs:** when running multiple servers together.
  + **Out of date or incorrect data replication:** when some sources may be out of sync due to any network failures, may lead to unnecessary data kept.

## Database Replication Implementation in Multi-users distributed Text Editor

We used ***MongoDB replica- sets*** for implementing the database replication technology in our Multi-Users Distributed Text Editor which is mainly characterized by being:

* Synchronous Replication
* Single leader Architecture based

And these types of database we have discussed in the previous section.

## Using MongoDB replica-sets

First, we implemented replication by creating 3 different databases on 3 different terminals on their own but all in the same replica set. Then, we defined one database of them as a PRIMARY database and the others are automatically defined as SECONDARY database.

We start by Running the command: *mongod -replSet Dist –dbpath mongos/db1 –port 27018*

To be able to configure our desired 3 database replicas in the same replica set.

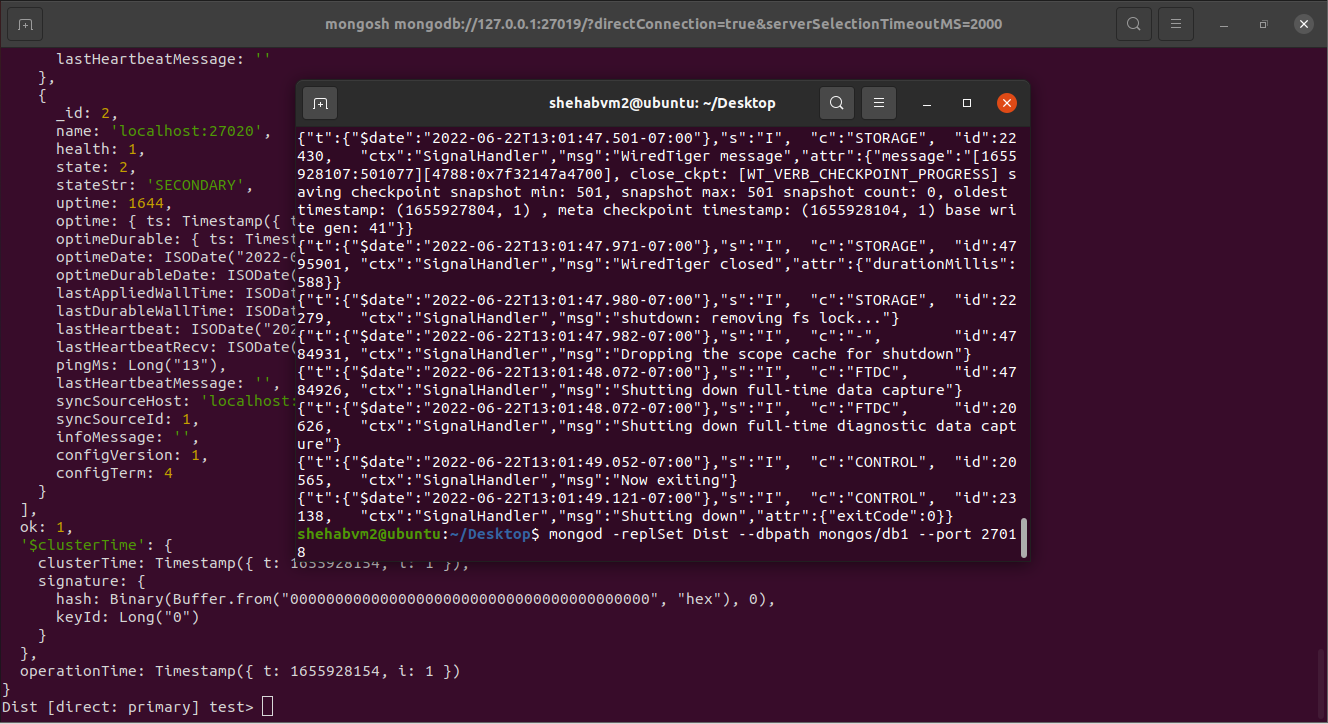


Figure 7: mongod command to start the MongoDB database associated with replicaSet

Then we start configuring our 3 database replicas by defining a variable *rsconfig*, where we define 3 databases with the same local host but different IDs and port numbers addresses of 27018, 27019, 27020. Later we initiate our replica set using the defined configuration variable *rsconfig* using *rs.initiate(rsconfig)*



Figure 8: configuration of the databases

Then we can check for the status of the replicaSet using the following command *rs.status()*

The members of the replica setwith the information for each database are then displayed, one database is specified as a PRIMARY database while the others are automatically set to secondary. each database has its own info of:

* ID
* Name “host + port number”
* State either primary, secondary, or unreachable

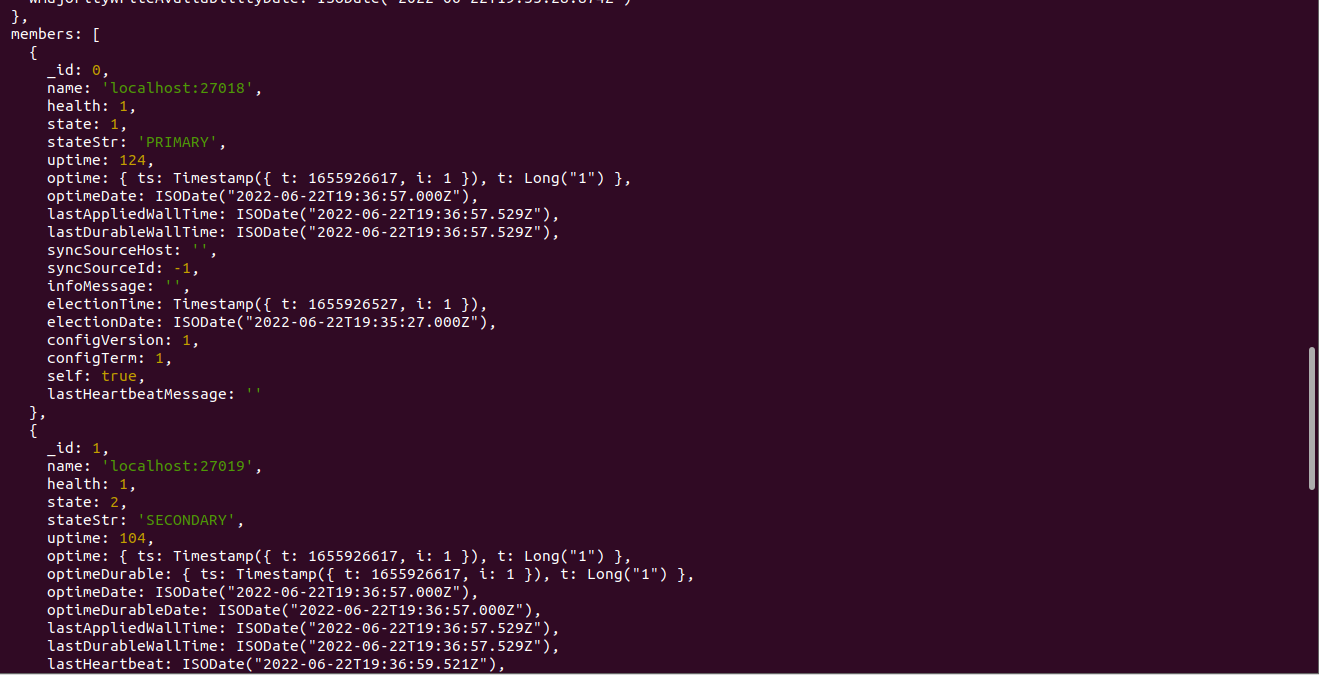


Figure 9: database 1 info

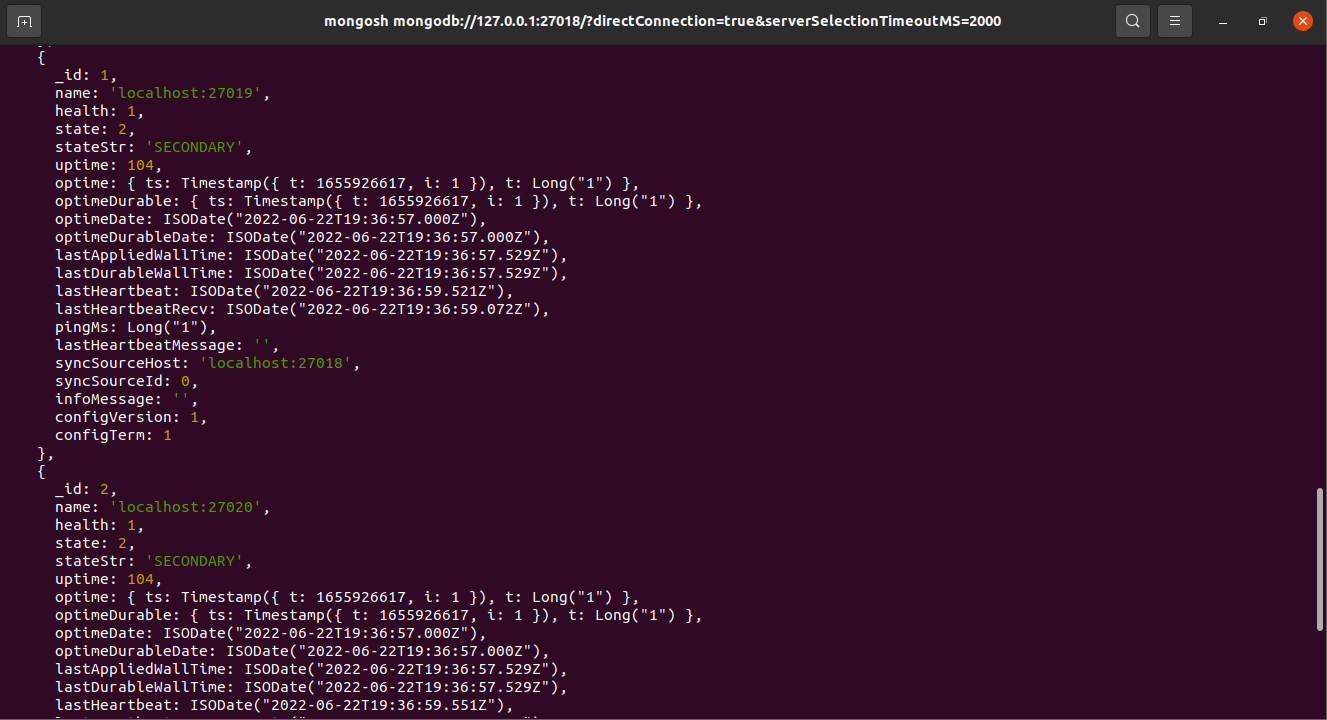


Figure 10: database 2 info

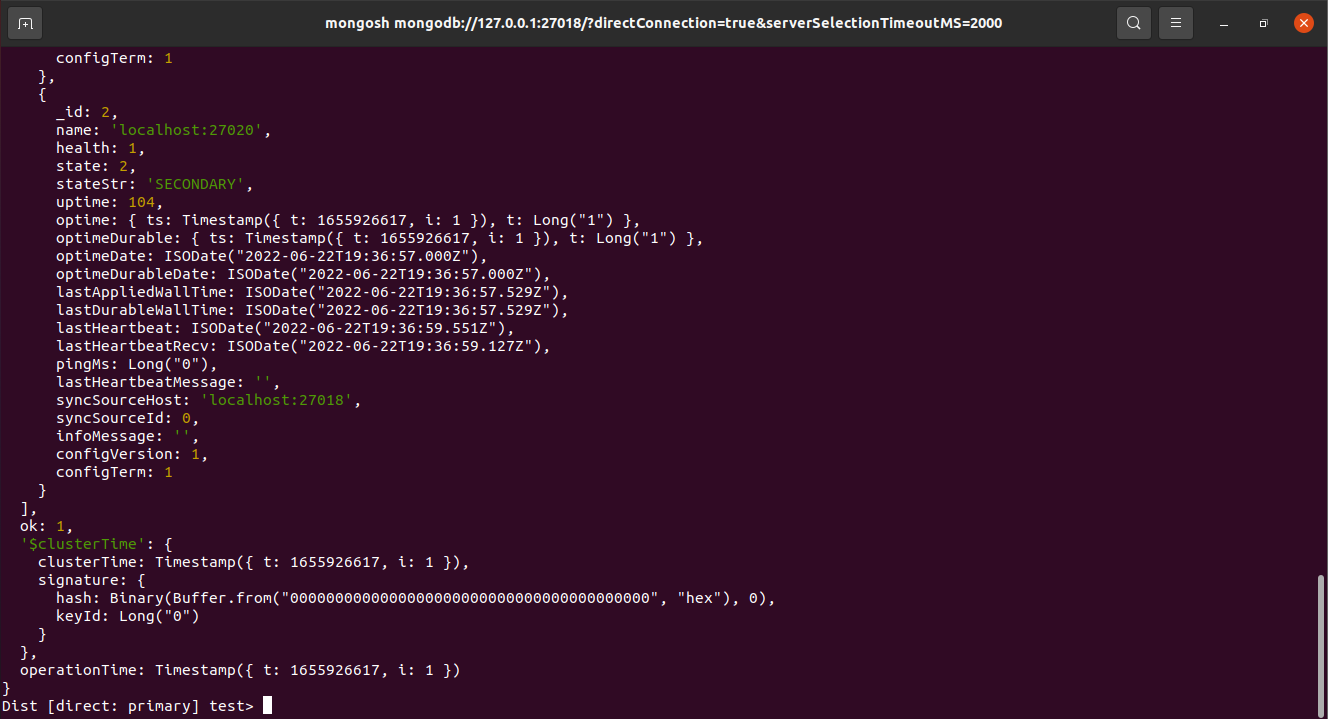


Figure 11: database 3 info

Then we tried to shut down Database 1 which has the port number 27018 and check which database will be the primary database. It was Database 2 which has the port number 27019 that became the new Primary database.

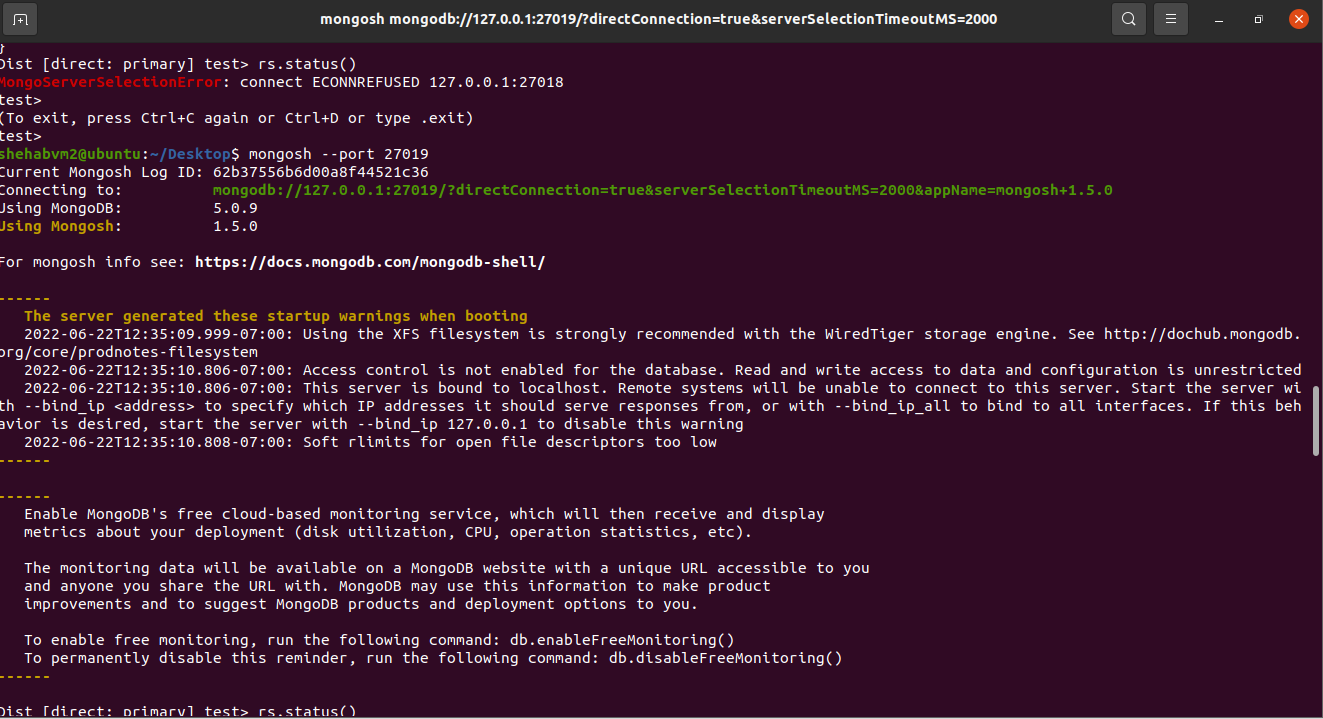


Figure 12: rs.Status() of DB1 cannot connect because of db failure.

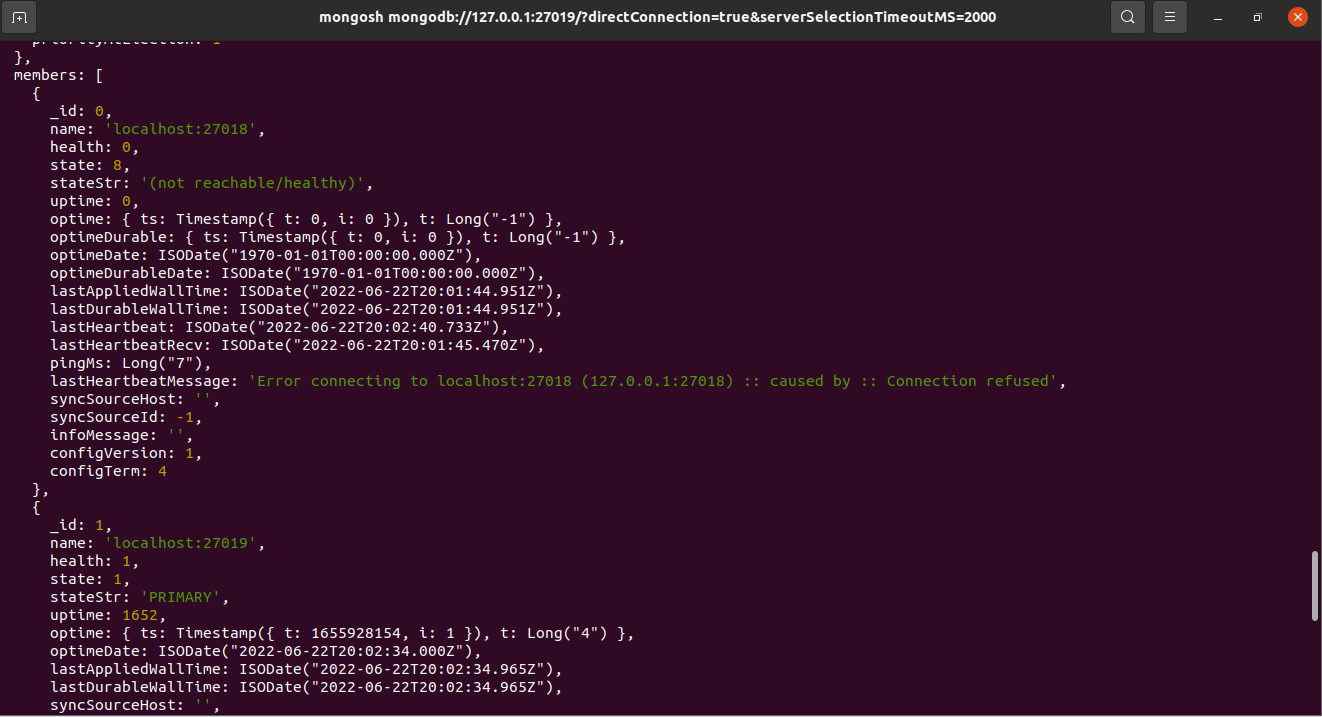


Figure 13: Database 1 became the Primary Database

As seen above, after shutting down database 0 with port number 27018. The database 1 with port number 27019 became the Primary database instead of being a Secondary database. While the third database 2 with port number 27020 stayed as a Secondary database. If we tried to bring back database 0 with port number 27018 and run it again. It will be a secondary database.

For the Primary database currently available, we are able to read and write to the collections, but for the secondary ones we are able only to read from it but an access is required to specify reading from the databases with status “Secondary”. This is done by running the command: *db.getMongo().setReadPref(‘secondary’).*

And to return the data written to the document in the primary to the users on the same shared document we use the command: *db.users.find()*

*Text

Description automatically generated*

Figure - Inserting data to a secondary database

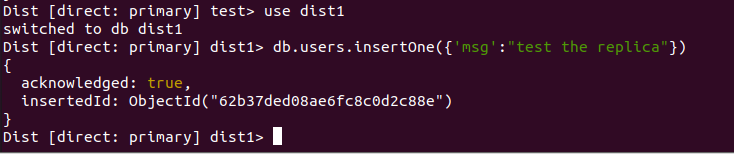


Figure 15: Inserting data on a Primary database successfully.



Figure 16:enabling read access on the secondary database

Text

Description automatically generated

Figure - Fetching documents from a secondary database.

## 

# High-Level System Design

# Testing Scenarios and Results:

# Failure scenarios:

# Conclusion:

# References

<https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk>

<https://softobiz.com/understanding-the-event-driven-architecture/>

<https://www.astera.com/type/blog/rest-api-definition/>

<https://restfulapi.net/>

<https://hevodata.com/learn/data-replication-in-distributed-system/#intro>