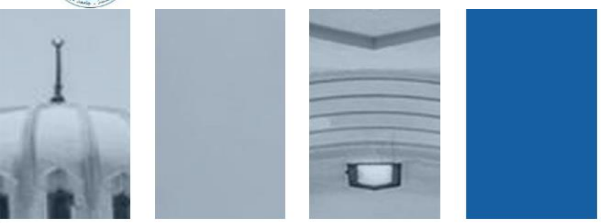

Program: Computer and Software Systems

Examination Committee:

Dr Ayman Bahaa

Eng: Mostafa Ashraf

Ain Shams University Faculty of Engineering

Spring 2022

Team Members:

1. **Shehab El Din Adel Nassef Attia - 18P3863**
2. **Zyad Ahmed Aly Hedar Yakan - 18P9538**
3. **Farah Amr Abd El Fattah – 18P3784**
4. **Eman Khaled Ahmed Ibrahim– 18P9713**

Table of Contents

Table of Figure

Table of tables

Code Snippets

I. 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.

II.Project Description

This project describes to design and implement the multi-user distributed text editor using different components to collaborate in reading and editing text documents. We firstly made the client interface code that will appear to the user using React.js, quill and SocketIO for communication with the server. Meanwhile, we implemented different servers to handle the clients' requests to edit or insert data. The servers are connected to with a MongoDB ReplicaSet. The implementation of the database required several databases, as there is primary database which allows read/write to it and there are 2 secondary databases which only allow reading to them, all of them are connected. If the primary database fails, an election occurs between the secondary databases, then one of them is selected to be the primary database. Moreover, we used Redis as a publish/subscribe architecture to allow WebSocket servers to communicate with each other as part of the scaling plan.

Also, we made a load balancer which will be replaced between the client and server which refers to the act of distributing network traffic across multiple services.

This makes sure that there is 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).

III. Beneficiaries of the project

This Multi-User Distributed text editor system can be used in various fields and aspects, and for various users with different backgrounds either technical or non-technical. The benefits of this system is shown in offering a user-friendly yet reliable interface for Shared Document/s between multiple users or editors. The concept of using shared documents between users is important as it reduces the time consumed in merging different versions among the multiple users enrolled in editing the same document, it also reduces the conflicts resulting from the clashes between different versions of the document.

Simply, it is better for all the enrolled users for editing a specific type of document to edit one version of a document with up-to-date edits of the other users all saved in database.

Beneficiaries of the project may appear clearly In any Organization of documenting important information which needs to be frequently updated by multiple users in various timings around different places (remotely or not), the system will be a perfect tool for handling all the requirements needed to maintain the functionalities required in the organization.

IV.Source Links

A. GitHub Repository Source Code

GitHub Repo

Name	Role
Eman Khaled	- Part of client Code - Project Documentation - Redis Publish Subscribe Implementation
Farah Amr Abd El Fattah	- Part of client Code - Project Documentation - Database Implementation
Shehab Eldin Adel	- Part of Server Code - Implementation Of Load Balancer - Redis Publish Subscribe Implementation - Database Replication Database Implementation
Zyad Ahmed Yakan	- Part of Server Code - Implementation Of Load Balancer - Database Implementation and Database Replication

B. Project Explanation Video Google Drive

Google Drive

V.Roles of the team

VI. Task Breakdown structure

C. Frontend

Implementation of Client Code by using Quill and web sockets offering an interface for the user to be able to write on the document by different tools and options .This is a pure example of and implement a clone of Google Documents application.

D. Backend

Implementation of Server Code by using SocketIO and Node.js.

E. Redis Publish Subscribe

We implemented Redis as Pub/sub messaging which can be used to enable communication between the servers.

F. Database

We used NoSQL MongoDB in order to persist the documents data, so that users can get back to where they left the documents.

G. Data base Replication

We implemented the database replication using MongoDB replica dataset, as if the primary database fails to work there are other secondary database that will take its place and work instead on it and become the new primary database. This replication dataset was implemented so that we can achieve data replication, availability, and concurrency. We implemented it using three running Docker containers on a DigitalOcean droplet.

H. 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). We wrote configurations using HaProxy and NGINX in order to make implement load balancing using roundrobin or ip-hashing algorithm.

VII.System Architecture

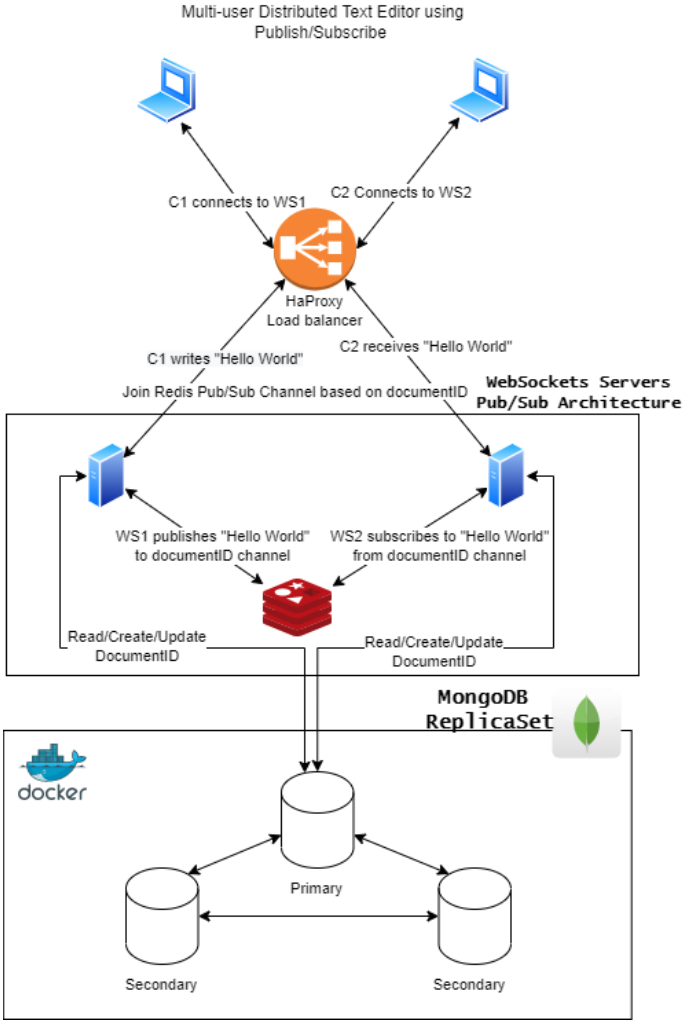

Figure 1 Multi-User Distributed Text Editor System Architecture

A. Architecture and Protocol Comparison

1. Web Sockets VS HTTP

Web Sockets

- 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 are not dynamic enough for those kinds of apps.
- Web Sockets 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.
- , which has several advantages. QUIC can prevent head-of-line blocking delays, improving network performance in many situations. This is a limitation of WebSocket.

2. RESTful vs Event-Driven Architectures

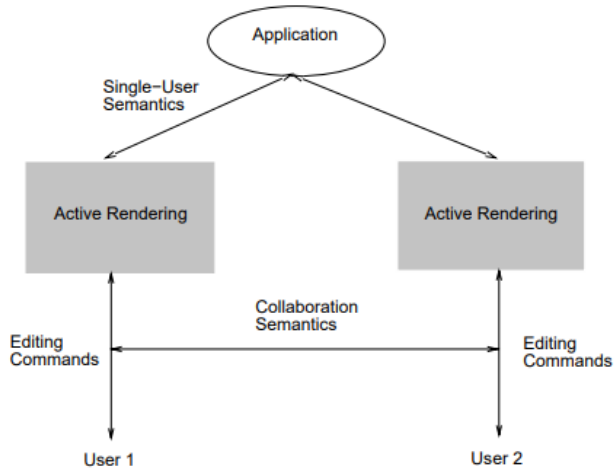

Figure 2 general architecture for collaborative editing apps

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. The View of 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. Those principles have greater **stability** because it restrains component performance. So that each component can't see further than the immediate layer with which it is intermingling.

Commented [11]: Paragraph instead of points

Commented [12R1]:

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. **Layered System**

   It 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.

3. **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
   - Self-descriptive messages
   - **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.
   - **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.

4. **Client-Server**

   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.

5. **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.

6. **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.

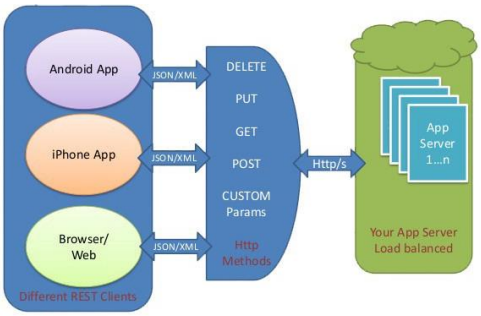

Figure 3 REST Architecture

- **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).

Event-Driven Architecture

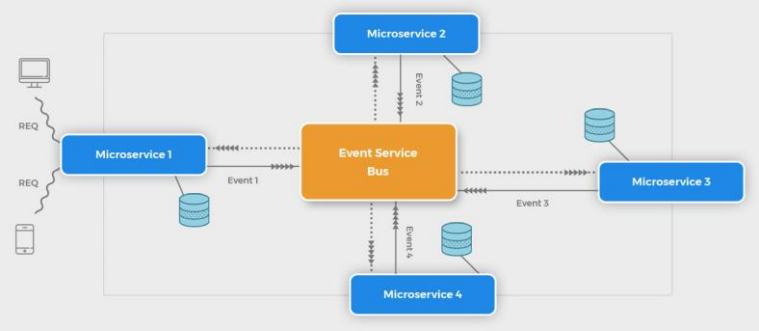

Figure 4 Event-Driven Architecture

- **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

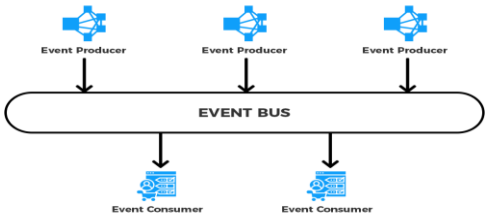

Figure 5 Event Channel Illustration

- **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:**
  1. Adding new events and processes is very easy in the event-driven architecture.
  2. Event-driven architecture also gives the transactional guarantee, getting notified of every successful transaction that occurs.
  3. The architecture is easily replaceable.
  4. We can roll-back changes or move to any event in the event-driven architecture. Which is helpful in case any issue occurs.
  5. 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.

VIII.System Design

A. 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.

B. 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 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.

C. 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.

D. 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

Whenever 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.

E. System Implementation

1. Client Implementation

We chose to use **quill** as a text editor since it provides us with almost all formatting options for the text editor. It also supports an important concept **delta,** which are operations that you can perform to get from one step to the next step as inserting characters in the page that you are writing in it. With every change, the **delta** operations will generate a string with the operation done, the character (if any) and the index of that operation. Ensuring that only one version of the document is available at all times with no conflicts, as it save their changes instead of saving the entire document. For the implementation of the connection between the host server and the clients we used **SocketIO** library.

```
export default function TextEditor() {
  const { id: id_doc } = useParams()
  const [newsocket, setSocket] = useState()
  const [quill, setQuill] = useState()

  console.log(id_doc)


  //UseEffect1
  useEffect(() => {
    //Setting a random name for the socket user connected
    const r = Math.floor(Math.random() * 10) + 1
    const s_socket = io("http://localhost:3001", { query: { username: `shehab ${r}` } })
    setSocket(s_socket)

    return () => {
      s_socket.disconnect()
    }

  }, [])
```

Code Snippet 1 Text Editor Function with Use effect () to connect with socket.

So here we are making the main function “Text Editor” and we are implementing quill inside of it and are using **use effect()** as we want to render one time. So firstly, **useeffect1()**, we are invoking the callback io and passing to it the URL the we are connecting to, by setting the port number to 3001 which is the server URL and will rename it as a socket. On the other side of the code, we are going to put the Client URL that we are going to discuss later.

```
//UseEffect3
useEffect(() => {
  if (newsocket == null || quill == null) return

  const socket_handler = (delta, oldDelta, source) => {
    if (source !== "user") return
    newsocket.emit("send-changes", delta)
  }
  quill.on("text-change", socket_handler)

  return () => {
    quill.off("text-change", socket_handler)
  }
}, [newsocket, quill])
```

Code Snippet 2 Changes Detection

And to finalize the socket, we are going to write the disconnect line at the end to make sure that the connection is disconnected after it was connected at the beginning. Now, the second **use Effect ()** will be used to be detecting changes whenever quill changes.

As it said, quill on text change runs this function as it passes us the delta, old delta and the source. The source is going to determine whether the user made these changes or whether the actual kill library made these changes so it is important to test to user that he made the changes first, else it is going to return and do nothing. As we only wanted to track the changes that the user made. Then we are going to send these changes by using sockets to the server to be saved and changed on our database by emitting a message from the client to the server and passing it the delta. This delta is very important regardless that it has to send the only parts changed not the whole document. Then at the end, we are going to remove the event listener that is no longer needed.

```
//UseEffect2
useEffect(() => {
  if (newsocket == null || quill == null) return    //check to make sure we have a socket adn a quill

  const socket_handler = delta => {
    quill.updateContents(delta)
  }
  newsocket.on("receive-changes", socket_handler)

  return () => {
    newsocket.off("receive-changes ", socket_handler)
  }
}, [newsocket, quill])
```

Code Snippet 3 Receiving Changes

Furthermore, the function of using the ***use Effect ()*** is used another time to do the operation of receive changes. Where we use the socket operation ***on ()*** to set up the event of *"receive-changes"* to our Server while the other parameter is a ***socket_handler*** to take in the delta that we get from our receive changes, That's why we used the ***updateContents ()*** feature from the quill tool to pass the delta which contain our last changes which is basically going to run these specific changes.

While the socket ***off()*** operation which takes the same parameters of *"receive-changes" and the Handler* will do exactly the same thing of setting up an event listener but this one is just updating our document to have the changes that are being passed from our other clients using the same document for editing.

By taking into considerations that clients using are two separate different computers working on the same document and our Server is doing all the Communication between the two objects (clients) to make sure the document is functioning exactly as we want.

```
const shortId = require('shortid')
function App() {
  return (
    <Router>
      <Switch>
        <Route path="/" exact>
          <Redirect to={`/documents/${shortId.generate()}`} />
        </Route>
        <Route path="/documents/:id">
          <TextEditor />
        </Route>
      </Switch>
    </Router>
  )
}
export default App
```

Code Snippet 4 React Routes

After setting up all the events for the server to listen to with using the ***use Effect ()*** instances and the quill key features functions, the clients here are modifying the same instance of document, all on local host: 3000. So we need to build a fix so we can then have multiple different documents by using ***React Router.***

After the imports and configurations, we use normal JavaScript switch cases with Wrapping Everything inside a Router Tag:

1. One Route for Redirecting to a new document by generating a unique id (***shortId***)
2. Second route for Rendering the text editor

```
//useEffect4
useEffect(() => {
  if (newsocket == null || quill == null) return

  newsocket.once("load-document", document_loaded => {
    quill.setContents(document_loaded)
    quill.enable()
  })

  newsocket.emit("get-document", id_doc)
}, [newsocket, quill, id_doc])
```

Code Snippet 5 Loading A Document

This ***use Effect ()*** is used after ensuring we have an access to the id we specified in the route from the URL of the document. What happens is every time the socket quill or the document Id changes we run the code in this ***use Effect ()*** event, if the socket or the quill is not null, we want to tell the server what the document is actually a part of. So we used the socket function ***emit ()*** and pass it *"get-document" & the document id.*

This function sends up to the server the document id so we can actually attach ourselves to the room for that document and if we have a document saved it is going to send us that

document back to us. All we have to do is just make sure we listen to that event. So we use the socket function ***once ()*** since we only want to listen to this event once, and this function will automatically clean up the event after it gets listened to once. This function takes *"load-document" and the document itself* as parameters.

So, when we call get document, it is going to pass up to our server our document, the server is going to do some background operations to load our document and finally it is going to send it back down to our client with this load document event.

Then we are going to use some quill functions ***setContents ()*** so we can load up our text editor and have the contents inside of it and ***enable ().*** And this is used because we want to disable our text editor until our document is loaded.

```
75  //useEffect5
76  useEffect(() => {
77    if (newsocket == null || quill == null) return
78
79    const time_interval = setInterval(() => {
80      newsocket.emit("save-document", quill.getContents())
81    }, interval_ms)
82
83    return () => {
84      clearInterval(time_interval)
85    }
86  }, [newsocket, quill])
```

Code Snippet 6 Saving data in DB

This use Effect () is basically setting a timer every couple of seconds we are going to save our document, this is done by the function ***setInterval ()*** and here we pass it how milliseconds we want to wait, and we also ***emit ()*** that save document function and in order to get the actual contents of our document, we use the ***quill.getContents()*** function to get all the information we need to save to the database. Clear interval so we don't run anymore

2. Server Implementation

Document DB Model

This file contains the declaration of database schema of the document we are using ***"DocumentSchema",*** with some attributes ***id, data, dateCreated and users*** which is of type of users' schema. Data is an object containing all the operations provided by QUILL features (insert, delete, etc.…).

Code Snippet 7 Document Model File

Database Connection File

It simply contains a function ***mongoose.connect()*** passing it the URL of the dB created to connect to the database replica set we created - *discussed in details in database replication section-* .

```
server > JS db.js > ...
1   const mongoose = require('mongoose')
2
3   const connect = async () => {
4       //Replace the database URI with process.env.DB_URL
5       //Which must be the MongoDB replicaSet URI
6       //const dbURL = `mongodb://${process.env.DB1},${process.env.DB2},${process.env.DB3}?replicaSet=Dist`
7       const dbURL= process.env.DB_URL
8       try {
9           mongoose.connect(dbURL)
10          console.log(`Connected to database successfully ${dbURL}`)
11      } catch (e) {
12          console.log(e)
13      }
14  }
15
16  module.exports.connect = connect
17
```

Code Snippet 8 Connecting to MongoDB replica set

Server Creation File

Starting by importing some libraries on the server side by importing the env file configurations that we are going to use later in the code by **require("dotend").config()** and the document model that we are previously explained by **require ("./model/document.model")** , A server is initialized by **require("socket.io")** and the Redis client by **require("redis")**. After that we are connecting to the MongoDB by using the function **dbconnect()**.

```
1   //Importing .env file configurations
2   //To be able to use .env variables
3   require("dotenv").config();
4   const dbConnect = require('./db').connect;
5   const { Server } = require('socket.io');
6   const Document = require('./model/Document.model');
7
8   const { createClient } = require('redis');
9   const { createAdapter } = require('@socket.io/redis-adapter');
10
11  //Connecting to the MongoDB database using
12  dbConnect()
13
```

Code Snippet 9 Importing of the Server.js File

Here we are setting up the server using environment that we were importing in the previous screenshot, by taking its **cors** which are the origin and the methods that the server can use which are post and get.

```
20
21  //Setup the server using environment
22  //variables for port number, and allowing
23  //CORS in order to allow access from
24  //The client code to our resources here.
25  const io = new Server(process.env.PORT, {
26      cors: {
27          origin: process.env.CLIENT_URL,
28          methods: ["GET", "POST"]
29      }
30  })
31
32
```

Code Snippet 10 Setup a new environment for the server

Here we are making a new variable that will be used several times which is **default Value** and initializing it with empty string. As it is the very first value in the document that will appear to the user in the first place.

```
34
35  //default value that is added to the document
36  //whenever we create a new document to be added
37  //to the database.
38  const defaultValue = ""
39
40
```

Code Snippet 11 Initializing the default Value of the Document

Here we are initializing some arrays. The first one is **Allclient []** which will contain all the clients that are establishing the web socket connection on the same server. So that the server could keep track of the number of clients connecting and disconnecting from it.

The second array is **redisPub** w **redisSub** are the arrays containing all the publisher and subscriber to the channel using the same web socket connection.

```
42  //Array contains all sockets that establish the connections
43  //with the server in order to keep track of number of clients
44  //connected and number of clients disconnected
45  var allClients = [];
46  var redisPub = []
47  var redisSub = []
48  /**
```

Code Snippet 12 Initializing some Arrays

This is the main **function createClient()** of creating the client by initializing the publisher and subscriber redis clients in order to subscribe to the same single channel so that all the websosckets can connected to each other using it. Also, when some changes happen, they all got to know the changes that happen or the request that arrives by single socket.

All of them are connected to the same URL first, then if this process works well, then the console will output a message that the publisher is ready to be used and connected successfully, else, will throw an error message.

Then after the successful connection, the clients will be pushed into the array all Clients by **redisPub.push (pubClient)** that we previously initialized so that the servers will have access to all the clients that are connected to it by the client socket ID.

```
54  //Initilizaed Publisher and Subscriber Redis Clients in order
55  //To subscribe to a single channel so that all websockets servers
56  //Can be connected to each other using it.
57  var pubClient = createClient({
58      url: `redis://:${process.env.REDIS_PASSWORD}@${process.env.REDIS_HOST}:${process.env.REDIS_PORT}`
59  });
60  pubClient.on('ready', () => {
61      console.log('Publisher connected to redis and ready to use')
62  })
63  pubClient.on('error', (err) => console.log('Publisher Client Error', err));
64
65  Promise.all([pubClient.connect()]).then(() => {
66      redisPub.push(pubClient)
67      console.log(`number of publishers is ${redisPub.length}`)
68  })
```

Code Snippet 13 CreateClient() function

This one is creating an event listener for the connection event upon request. This will happen and run whenever one client establish the connection with the server and the server is ready to connect with it upon the channel.

So Firstly, the connection is establish and a message is printed on the console by **console.log ("subscriber is connected to redis and ready to be used")** for the server that the client which acts as a subscriber to the redis channel that it is connected successfully and ready to be use

Then when the subscriber are connecting successfully, it returned to the client that he is connected on which server with the username of the server that the client is connecting on it right now by **console.log ("subscriber connected to $(username))**

Then after the successful connection, the clients will be pushed into the array allclients by **redisPub.push(pubClient)** that we previously initializing so that the servers will have access to all the clients that are connected to it by the client socket ID.

```
70  //Creating event listener for connection event
71  //That is listened to whenever a client establishes
72  //A connection with the server
73
74  io.on("connection", (socket) => {
75
76      const subClient = pubClient.duplicate();
77      subClient.on('ready', () => {
78          console.log('Subscriber connected to redis and ready to use')
79      })
80      subClient.on('error', (err) => console.log('Subscriber Client Error', err));
81
82      Promise.all([subClient.connect()]).then(() => {
83          //Connecting the socket server to the redis channel
84          //using Socket.io Redis-Adapter
85          console.log(`A Subscriber clients connected ${username}`)
86          redisSub.push(subClient);
87          console.log(`number of subscribers is ${redisSub.length}`)
88      });
89
```

Code Snippet 14 Create the connection Socket

Here is to whenever the client connected to the server, we fetch the server username from the **socket.handshake.querey** library so that we can log it with the username on it.

Then the socket ID will be pushed to the **allclients** Array by **allClients.push (socket)** that we previously initialized so that the server will get to access the sockets that are connected and subscribed to the same redis channel.

```
89
90      //Whenever a client connects to a server
91      //We fetch his username from the handshake.query
92      //and log it
93      allClients.push(socket)
94      var username = socket.handshake.query.username
95      console.log(`A client is connected! ${username} - Number of sockets is: ${allClients.length}`)
96
```

Code Snippet 15 Push function in the server side

For the next snippets of code, these are the events that the sever listens to. We have stated that the number of sockets represents the number of clients or users using or editing the document currently and these sockets do subscribe on the channel according to the document id they are currently using.

1st event is: disconnecting the sender client from the connection associated by unsubscribing it from the redis channel so that when the document or data is returned to the rest of clients' sockets, it will not be redundant at the same user who sent this data.

That is why the un-subscription from the channel after sending or publishing is important. Also, because the channel may have maximum number of subscribers to the channel, so if this subscription is not considered, this limit of max subscribers may be reached but with actually no clients using the document or alive on connection currently but they did not unsubscribe from channel just after sending their data of changes on documents.

```
 99     //Event listener for client's socket disconnect
100     //Event that listens to any
101     socket.on('disconnect', async function (reason) {
102         //Unsubscribe from the redis channel
103         console.log(`${username} got disconnected due to ${reason}`)
104         var i = allClients.indexOf(socket);
105         allClients.splice(i, 1);
106         console.log(`Number of sockets now is: ${allClients.length}`)
107         var subI = redisSub.indexOf(subClient)
108         redisSub.splice(subI, 1)
109         //Unsubscribe from all the channels
110         await subClient.unsubscribe()
111         await subClient.quit()
112     })
```

Code Snippet 16 Disconnecting & unsubscribing Event

2nd event is: getting the document as when receiving the data, the subscriber do **parsing of messages** sent by the publisher to return to its form.

Then the ***emit ()*** function is executed to receive the document with the current changes that is updated by users. The sockets of users editing the same document must join the same room specified by the document Id, this is implemented by the ***LookUpDocument ()*** function passing it the id of the document which is being edited.

The most important part in this event, is checking that socket id of the sender of changes is not being emitted the document to, so that to not have a redundant updates or data in his version of document, so the document is only emitted to the rest of sockets in the room excepting the sender client's socket.

```
114
115     socket.on('get-document', async (documentID) => {
116         try {
117             //when receiving as a subscriber pare the data sent by the publisher to return to its form
118             await subClient.subscribe(documentID, (message) => {
119                 const msg = JSON.parse(message)
120                 console.log(msg.sender)
121                 console.log(msg.data)
122                 console.log(username)
123                 if (socket.id !== msg.sender) {
124                     socket.emit('receive-changes', msg.data)
125                 }
126             })
127         } catch (error) {
128             console.error(error)
129         }
130         const document = await lookUpDocument(documentID);
131         //TODO subscribe the socket to redis channel using the documentID
132         socket.join(documentID);
133         socket.emit("load-document", document.data); //on load/reload
```

Code Snippet 17 Get Document Event

3rd event is: when sending any changes, the server get the delta (which is a variable declared as the data of the document being edited) and change it to string when publishing occurs (to be accepted by JSON).

So the server or publisher will publish data by getting the Document ID and the sent messages which was just parsed to the subscribers.

```
134     socket.on("send-changes", async (delta) => {
135         //Change delta to string when publishing
136         //socket.to(documentID).emit("receive-changes", delta)
137         try {
138             const message = {
139                 'sender': socket.id,
140                 'data': delta
141             }
142             const sentMsg = JSON.stringify(message)
143             await pubClient.publish(documentID, sentMsg)
144             console.log(`${username} published`)
145         } catch (error) {
146             console.error(error)
147         }
148     })
```

Code Snippet 18 Sending changes of data event

4th event is: saving the document, which is basically getting the document and saving the data Up to Date to the database built using the function ***findByIdAndUpdate()*** and passing the document Id we want to save.

```
150     socket.on("save-document", async (data) => {
151         try {
152             await Document.findByIdAndUpdate(documentID, { data })
153         } catch (e) {
154             console.log(e)
155         }
156     })
```

Code Snippet 19 Saving Document event

5th event is: committing history on the socket which simply adds the commit history to the database based on date for example.

Here is to the **lookupDocument()** function that take the socket ID as a parameter.
It firstly checks if the parameter is Null to return nothing, else to check the document with this ID by **await Document.findbyid(id)** in the database, if found then it will return the whole document to the client as a return message. Else if the ID is not found, then it will create a new document with this new ID and will put **the initialization value = default Value** which we described earlier in the code which is an empty string, to open an empty document for the user.

```
67  async function lookUpDocument(id) {
68      if (id == null) return
69
70      try {
71          const document = await Document.findById(id)
72          if (document) return document
73          return await Document.create({ _id: id, data: defaultValue })
74      } catch (e) {
75          console.log(e)
76      }
77  }
78
```

Code Snippet 20 Look up document function

```
115  socket.on('get-document', async (documentID) => {
116      try {
117          //when receiving as a subscriber pare the data sent by the publisher to return to its form
118          await subClient.subscribe(documentID, (message) => {
119              const msg = JSON.parse(message)
120              console.log(msg.sender)
121              console.log(msg.data)
122              console.log(username)
123              if (socket.id !== msg.sender) {
124                  socket.emit('receive-changes', msg.data)
125              }
126          })
127      } catch (error) {
128          console.error(error)
129      }
130      const document = await lookUpDocument(documentID);
131      //TODO subscribe the socket to redis channel using the documentID
132      socket.join(documentID);
133      socket.emit("load-document", document.data); //on load/reload
134      socket.on("send-changes", async (delta) => {
135          //Change delta to string when publishing
136          //socket.to(documentID).emit("receive-changes", delta)
137          try {
138              const message = {
139                  'sender': socket.id,
140                  'data': delta
141              }
142              const sentMsg = JSON.stringify(message)
143              await pubClient.publish(documentID, sentMsg)
144              console.log(`${username} published`)
145          } catch (error) {
146              console.error(error)
147          }
148      })
```

Code Snippet 21 Redis Publish Subscribe Architecture

First, all the servers started to subscribe to the redis channel in order for all edits to be sent to all of them. This subscription takes the document channel ID and the data or message to be sent to all of the servers.

Then there is a check in line 123-124 for the socket number with the one who sent the data with, as the data will be sent to all the sockets except the one who sent among this message with.

Noted that the servers are acting as a publisher and subscriber both to the same channel. So the sockets now are connected to the servers, so when one server send a request (by the socket ID and the data wanted to be send or edits in the document) , then these messages will be sent to all sockets that are connected to the servers except the one who send the message.

```
149
150         socket.on("save-document", async (data) => {
151             try {
152                 await Document.findByIdAndUpdate(documentID, { data })
153             } catch (e) {
154                 console.log(e)
155             }
156         })
157         //TODO Group the last 3 minutes of changes into one 'commit'
158         socket.on('commit-history', async (data) => {
159             //Add the commit history to the database based on current
160             //Date for example
161         })
162
163     })
164 })
165
```

Code Snippet 22 Redis Publish Subscribe commit to the database

Here, sockets are waiting for any changes to be received else, there are going to commit the history to the database based on what message received or changes happen.

```
115     socket.on('get-document', async (documentID) => {
116         try {
117             //when receiving as a subscriber pare the data sent by the publisher to return to its form
118             await subClient.subscribe(documentID, (message) => {
119                 const msg = JSON.parse(message)
120                 console.log(msg.sender)
121                 console.log(msg.data)
122                 console.log(username)
123                 if (socket.id !== msg.sender) {
124                     socket.emit('receive-changes', msg.data)
125                 }
126             })
127         } catch (error) {
128             console.error(error)
129         }
130         const document = await lookUpDocument(documentID);
131         //TODO subscribe the socket to redis channel using the documentID
132         socket.join(documentID);
133         socket.emit("load-document", document.data); //on load/reload
134         socket.on("send-changes", async (delta) => {
135             //Change delta to string when publishing
136             //socket.to(documentID).emit("receive-changes", delta)
137             try {
138                 const message = {
139                     'sender': socket.id,
140                     'data': delta
141                 }
142                 const sentMsg = JSON.stringify(message)
143                 await pubClient.publish(documentID, sentMsg)
144                 console.log(`${username} published`)
145             } catch (error) {
146                 console.error(error)
147             }
148         })
```

Code Snippet 23 Redis Publish Subscribe Architecture

First, all the servers started to subscribe to the redis channel in order for all edits to be sent to all of them. This subscription takes the document channel ID and the data or message to be sent to all of the servers.

Then there is a check in line 123-124 for the socket number with the one who sent the data with, as the data will be sent to all the sockets except the one who sent among this message with.

Noted that the servers are acting as a publisher and subscriber both to the same channel. So the sockets now are connected to the servers, so when one server send a request (by the socket ID and the data wanted to be send or edits in the document) , then these messages will be sent to all sockets that are connected to the servers except the one who send the message.

```
49
50          socket.on("save-document", async (data) => {
51              try {
52                  await Document.findByIdAndUpdate(documentID, { data })
53              } catch (e) {
54                  console.log(e)
55              }
56          })
57          //TODO Group the last 3 minutes of changes into one 'commit'
58          socket.on('commit-history', async (data) => {
59              //Add the commit history to the database based on current
60              //Date for example
61          })
62
63      })
64  })
65
```

Code Snippet 24 Redis Publish Subscribe commit to the database

Here, sockets are waiting for any changes to be received else, there are going to commit the history to the database based on what message received or changes happen.

3. 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 are not being subjected to loads beyond what they are capable of withstanding.
- This is done to ensure maximum speed and capacity utilization

Load Balancer Implementation

In the first approach, we tried to implement load balancing using HaProxy, where we had to get a domain that points to a DigitalOcean droplet, then we had to generate an SSL certificate in order to allow transporting connections from the frontend to the backend securely. We wrote the following configurations

```
frontend distFE
         bind *:80
        bind *:443 ssl crt /haproxy.pem
         timeout client 1000s
         mode http
         default_backend distBE

backend distBE
         mode http
         timeout server 1000s
         timeout connect 1000s
         server ws1 https://dist-ws1.herokuapp.com
         server ws2 https://dist-ws2.herokuapp.com
```

Here we bind our server to listen on port numbers 80, and 443 for secure connections. We have a client timeout of 1000s, and http to allow http requests.
Then we have listed the backend servers addresses ws1, and ws2 in order route our connections from the frontend to the backend servers.

Yet we had several problems with HaProxy, so we had to switch NGINX. In order to upgrade the request from http protocol. We wrote the following configurations to implement load balancing using NGINX. Also, we used IP hashing algorithm instead of Round robin. IP hashing method instead. IP hashing uses the visitors IP address as a key to determine which host should be selected to service the request. This allows the visitors to be each time directed to the same server, granted that the server is available and the visitor's IP address hasn't changed.

Check this GitHub issue regarding the whole approaches taken

```
# Define which servers to include in the load balancing scheme.
# It's best to use the servers' private IPs for better performance and security.
http {
   upstream ws {
             ip_hash;
             server dist-ws1.herokuapp.com;
             server dist-ws2.herokuapp.com;
          }

   # This server accepts all traffic to port 80 and passes it to the upstream.
   # Notice that the upstream name and the proxy_pass need to match.

   server {
         listen 80;
         server_name disthaproxy.ddns.net;

      location / {
       #Upgrading the connection in order to establish the websocket connection
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_http_version 1.1;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_pass http://ws;
    }
   }

   server {
   listen 443 ssl;
   server_name disthaproxy.ddns.net;
   ssl_certificate #/etc/letsencrypt/live/distproxy.ddns.net/cert.pem;
   ssl_certificate_key /etc/letsencrypt/live/distproxy.ddns.net/privkey.pem;
   ssl_protocols TLSv1 TLSv1.1 TLSv1.2;

   location / {
        #Upgrading the connection in order to establish the websocket connection
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_http_version 1.1;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_pass http://ws;
    }
}

}
```

4. Database Replication

Due to the crucial component in the distributed systems applications which 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.

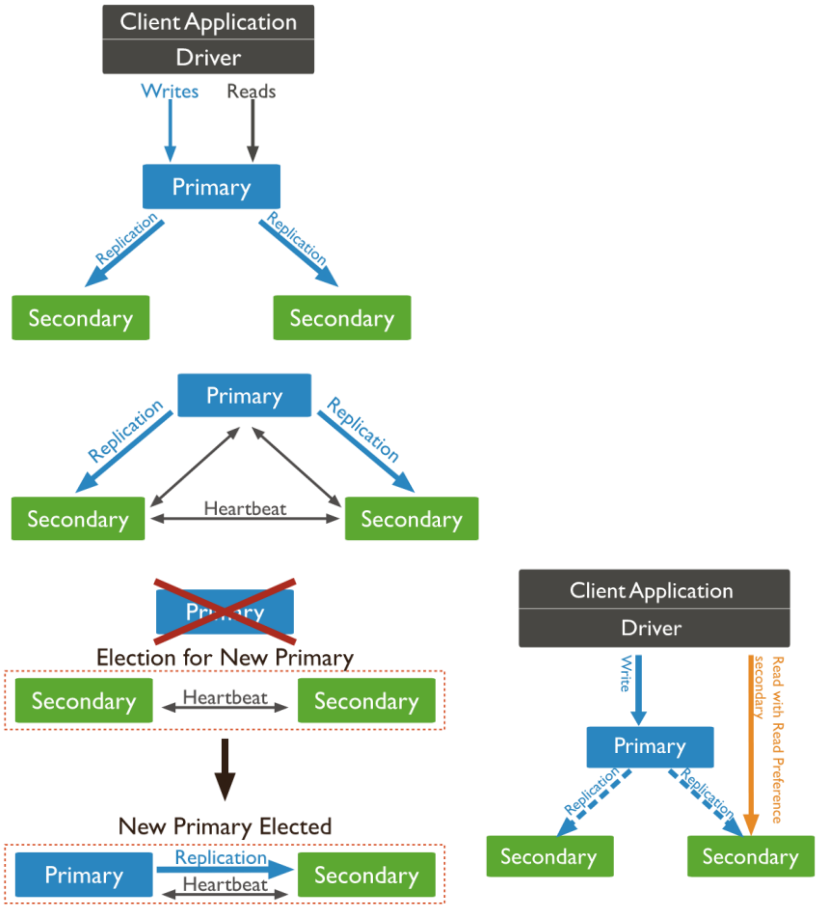

Figure 6 Database replication illustration

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

Table 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.

Table 2 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 replicaSets*** 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

We used Docker in order to run three containers in a Replica Set called `Dist` which allowed us to run MongoDB processes in the background.

```
sudo docker run -d -p 30001:27017 --net DistN --name db1 mongo:latest --replSet Dist
sudo docker run -d -p 30002:27017 --net DistN --name db2 mongo:latest --replSet Dist
sudo docker run -d -p 30003:27017 --net DistN --name db3 mongo:latest --replSet Dist
```

Then we were able to connect to one of them using MongoDB Shell, and initialize our ReplicaSet using the following configuration. Check the following GitHub issue for more details about our approach

```
rsconfig={_id:"Dist",
members:[
{_id:0, host:"104.248.130.64:30001"},
{_id:1, host:"104.248.130.64:30002"},
{_id:2, host:"104.248.130.64:30003"}
]}
```

IX. Testing Scenarios and Results:

A. Failure scenarios:

1. **When Primary database fails →** the mongo DB replica sets switches automatically to a secondary database that will take the place of the Primary one so no data loss occurs and the processes of the system remain working; shown in the following figures.

```
members: [
  {
    _id: 0,
    name: '104.248.130.64:30001',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    uptime: 142,
    optime: [Object],
    optimeDurable: [Object],
    optimeDate: 2022-06-26T00:55:58.000Z,
    optimeDurableDate: 2022-06-26T00:55:58.000Z,
    lastAppliedWallTime: 2022-06-26T00:55:58.390Z,
    lastDurableWallTime: 2022-06-26T00:55:58.390Z,
    lastHeartbeat: 2022-06-26T00:55:58.494Z,
    lastHeartbeatRecv: 2022-06-26T00:55:57.603Z,
    pingMs: Long("0"),
    lastHeartbeatMessage: '',
    syncSourceHost: '104.248.130.64:30003',
    syncSourceId: 2,
    infoMessage: '',
    configVersion: 1,
    configTerm: 2
  },
```


Figure 7: database 1

```
  {
    _id: 1,
    name: '104.248.130.64:30002',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 1439,
    optime: [Object],
    optimeDate: 2022-06-26T00:55:58.000Z,
    lastAppliedWallTime: 2022-06-26T00:55:58.390Z,
    lastDurableWallTime: 2022-06-26T00:55:58.390Z,
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: '',
    electionTime: Timestamp({ t: 1656204578, i: 1 }),
    electionDate: 2022-06-26T00:49:38.000Z,
    configVersion: 1,
    configTerm: 2,
    self: true,
    lastHeartbeatMessage: ''
  },
```


Figure 8: database 2

Figure 9: database 3

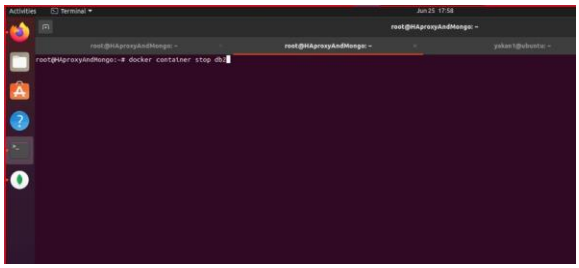

Figure 10: stopping current primary

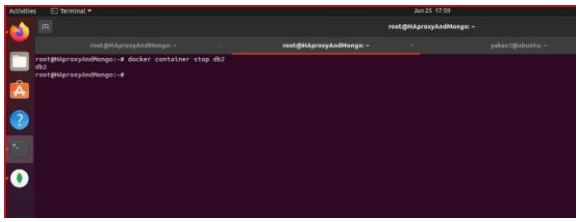

Figure 11: Primary database stopped

Figure 12: changed database after terminating the primary

Figure 13: Terminated Server

Figure 14: third database

```
root@HAproxyAndMongo:~# docker container start db2
db2
root@HAproxyAndMongo:~#
```

Figure 15: restarted previous primary container

```
members: [
  {
    _id: 0,
    name: '104.248.130.64:30001',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 873,
    optime: [Object],
    optimeDate: 2022-06-26T01:08:02.000Z,
    lastAppliedWallTime: 2022-06-26T01:08:02.072Z,
    lastDurableWallTime: 2022-06-26T01:08:02.072Z,
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: '',
    electionTime: Timestamp({ t: 1656205142, i: 1 }),
    electionDate: 2022-06-26T00:59:02.000Z,
    configVersion: 1,
    configTerm: 3,
    self: true,
    lastHeartbeatMessage: ''
  },
```

Figure 16: After Restarting

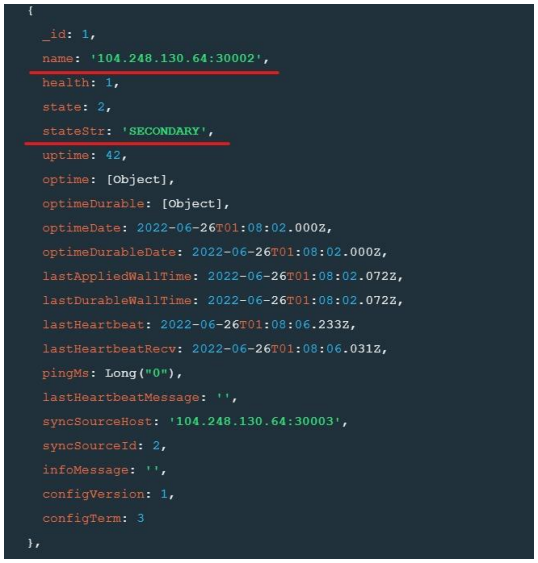

Figure 17: previous Primary database now restarted as Secondary

Figure 18: Secondary database currently

2. **When the client's connection is lost →** the client is blocked waiting for the connection to return. As the connection is lost between the client and the server, so the client cannot

insert new data on the screen to be saved on the database. So the client must wait for the connection to return in order to establish a secure web socket to the server, so that when he requests to insert or make any new changes that will make a difference in the database, there will be a connection between the server and the database to insert these changes on the database without any failures or loss of Data.

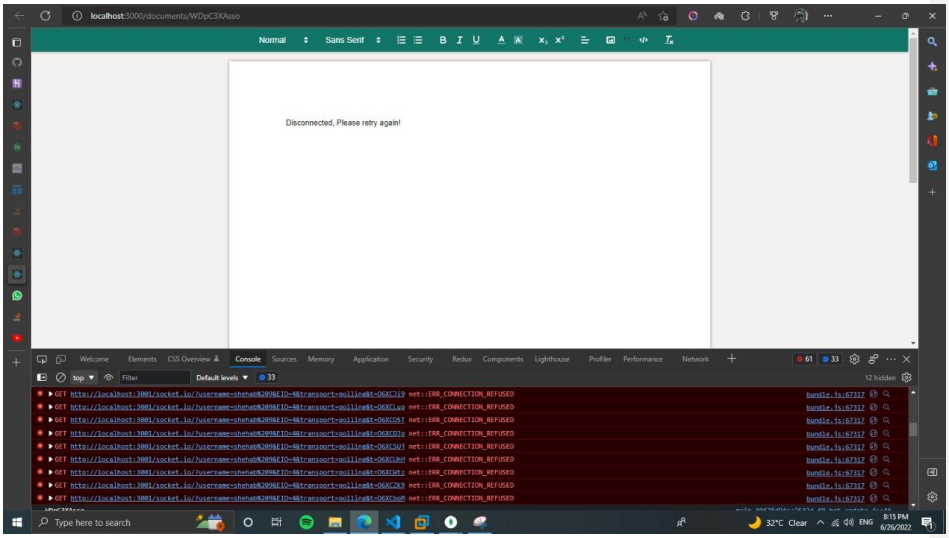

Figure 19: when client's connection is lost

X.Conclusion

The purpose of this project is to design and implement the multi-user distributed text editor using different components to collaborate in reading and editing text documents. Based on the analysis conveyed, it can be concluded that .The concept of using shared documents between users is important as it reduces the time consumed in merging different versions among the multiple users enrolled in editing the same document, it also reduces the conflicts resulting from the clashes between different versions of the document. As this clone of Google document can help many different people through different fields for writing, reading and editing a text document so that all changes can appear in the real time to all clients that using the same document. It's Crystal clear that this project has many benefits for technical and the non-technical people as the system will be a perfect tool for handling all the requirements needed to maintain the functionalities required in the organization.

XI.End User Guide

Opening the URL with the local host:3000 we work on, the document interface is displayed where the users are provided with all the tools needed to write and edit on the document.
These features are all offered from **QUILL** Text editor which we used to implement the front-end of the system – *discussed in the client code in the event-driven implementation section* – providing the interface of the document to be edited.

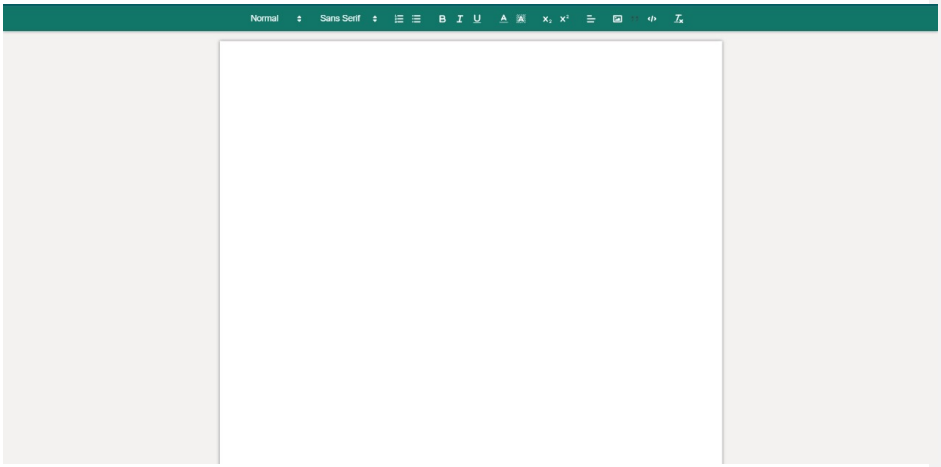

Figure 20: document interface

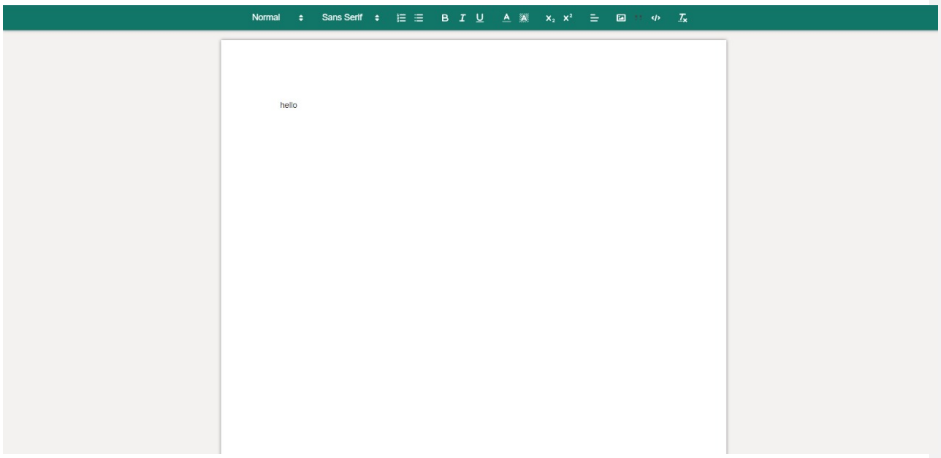

Figure 21: user one starts editing the document

Another user who want to edit or add/remove updates to/from the document will open the document on the same Local host which appears in the URL when he opens the same document *– the users on the same document are all joined a room to the channel connecting the servers (Redis) and connected to the same document by document ID; discussed in the server code in the event-driven implementation section* – then he will be able to receive the last updated document retrieved from the database, do he edits and then the document he updated will be saved to the database too for further retrievals.

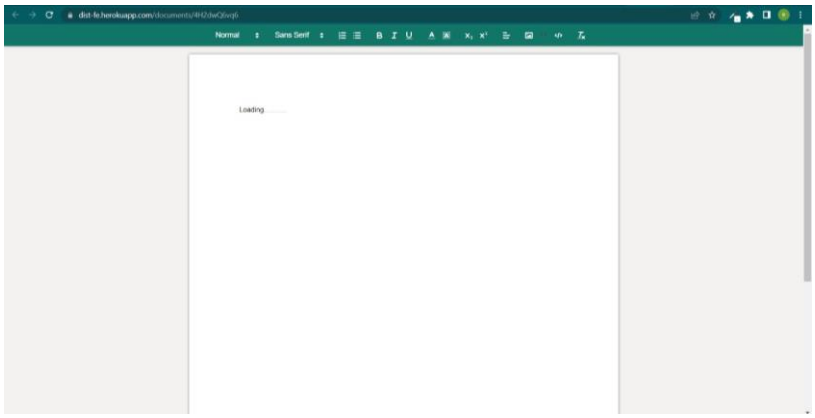

Figure 22: other user 2 opening the document waiting for updates to be retrieved

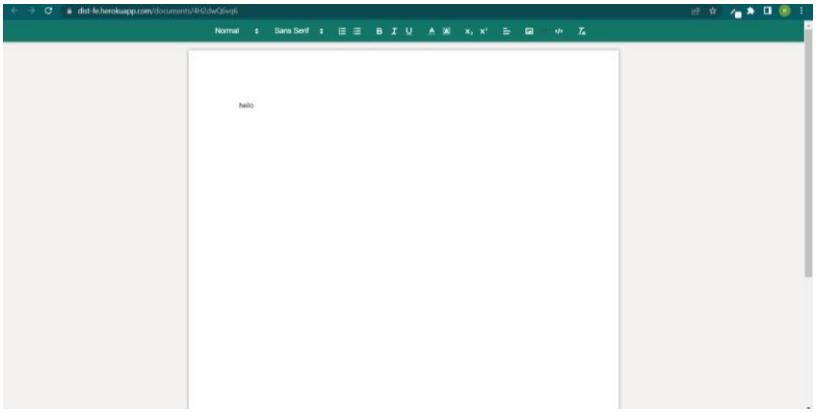

Figure 23: user 2 got the updated document successfully

User can be able to use various tools for editing the document as he desires then the document is automatically saved to the database - *replication is also implemented using mongoDb replica sets; discussed in detail in the database replication section* - also other multiple users can be able to get this version and re edit on it.

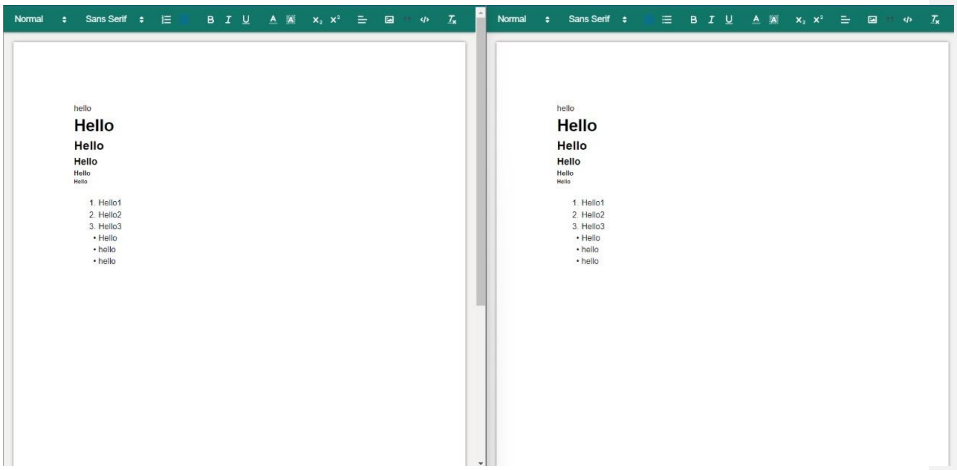

Figure 24: 2 parallel users editing the document with handling updates and using various tools provided for editing process.

XII.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