Proof of Concept: Real-Time Chat Application (Group 13)

Background

Using Websocket technology, we can build a real-time chat application which will allow users to communicate with low latency. The bidirectional nature of WebSocket connections allows for communication of live events between the server and the clients.

This document outlines the proposed Proof of Concept for a real-time chat application using WebSocket for low-latency and directional communication between clients and servers

Motivation and Objective

The motivation of the chat application is to offer seamless real-time communication while being scalable, fault-tolerant, and decentralized. The objective is to provide users with a reliable platform for one-to-one communication, with a focus on responsive user experience. In today's connected world, users expect uninterrupted access to communication systems. Therefore, our distributed chat application will work on high availability, reduce latency to enhance user experience, manage load using load balancers, and enhance security and protect data integrity by decentralizing data storage and transmission.

By using a distributed architecture we hope to overcome the limitation of centralized systems and enhance the overall reliability and performance of the chat application as it is not dependent on a single server or location.

The application aims to provide:

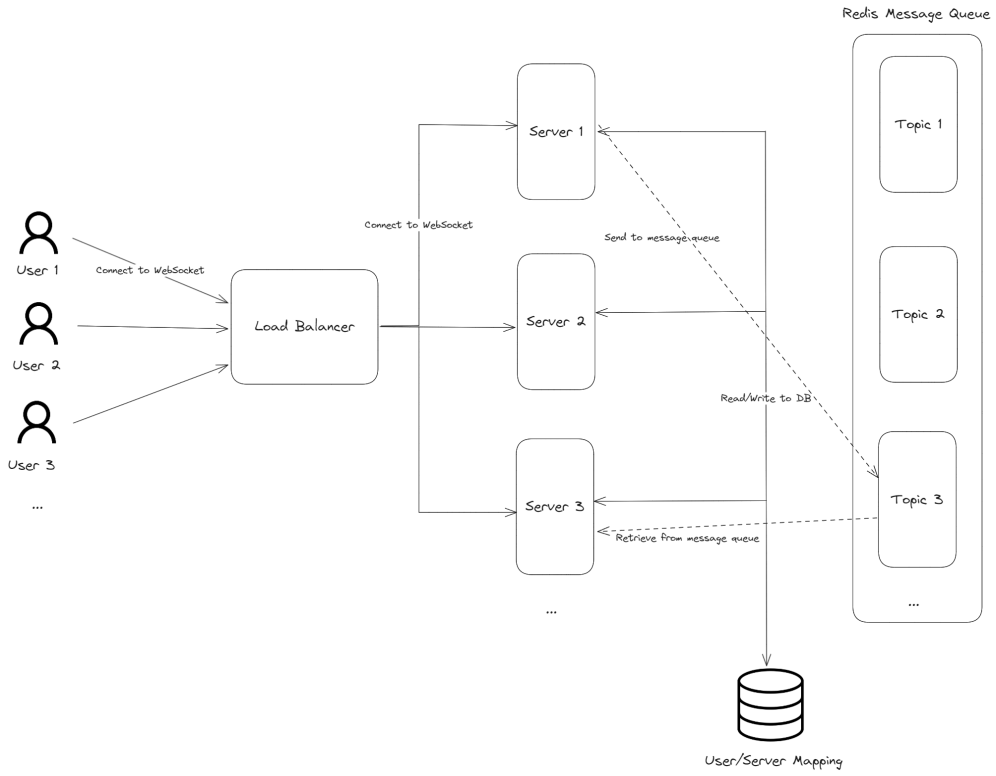
- Seamless real-time communication
- Scalability, fault tolerance and decentralization
- High availability
- Reduced latency, promoting enhanced user experience
- Load balancing for work distribution
- Enhanced Security and data integrity through decentralized data storage and transmission

Overview of the Application

The application would be focused on one-on-one chatting between users, where the users may be connected to different servers through the load balancer. Despite this, communication between the users will still be established. Using load balancers would efficiently distribute workload among servers and help develop mechanisms to handle server failures.

High-level architecture (Client/Server, Peer-to-Peer, or Hybrid); a diagram here can be helpful

The High-level architecture we will be using is the Client/server architecture. The Client/Server architecture will consist of the load balancer and the WebSocket servers. The servers are able to input and retrieve from message queues. Each user sends the message to their respective server from which the respective server must use the User Mapping table to determine which server the receiving user is connected to. Alongside this the sender's server will send the message to the receiver server's topic in the redis queue. The message can now be retrieved by the receivers server and be sent to the receiving user.
server 0



Discuss how you intend to implement: replication, communication, synchronization, consistency, and fault tolerance

Internal Synchronization is what we plan to use, this is because external synchronization creates a single point of failure and does not have good scalability as more servers try to request time from it, thus making it a poor fit. As internal synchronization would be based on calculations done from the servers themselves, this would distribute the chances of the synched time being off by a noticeable amount.

In terms of communication, the message-queuing model (message oriented persistent communication) is what will be used. Since Redis offers a Message Queue that has very fast read and write times, this becomes an optimal solution for communication between the server instances. Redis gives an interface to the server instances to retrieve messages from their respective Topic (queue), or to put messages into another server's Topic(queue). Each server will have its respective Topic in the Redis message queue. Redis acts as the message broker in the message-queuing model.

The handling of consistency could either be approached from a sequential or causal standpoint where either provide sequencing for related messages, causal has the difference of not requiring sequencing for everything that could provide more flexibility to the system.

Fault Tolerance:

A decentralized architecture introduces several failure modes that need to be addressed. It is necessary to make sure that the connection is handled robustly to manage any kind of disruptions in network connectivity. This failure can be kept in check by using non-blocking sockets or asynchronous I/O to handle multiple connections. It is also important to have an error handling mechanism in place to manage many possible network errors - for example, unlikely refused connections or disconnections. Having try-catch blocks in code is a useful way to catch exceptions and handle errors. It is always a benefit to ensure that the system is scalable and can handle message routings and delivery. Using acknowledgements and retrying to send messages in case of failures is a good way to help cover for this failure. Preventing data leaks by implementing TSL/SSL for secure connection over sockets and managing peer lists dynamically for better connectivity of peers are very helpful for security and integrity.

We will work to incorporate above mentioned ideas and even more for maximum security, reliability and scalability features to make sure our system is fault tolerant. We also plan to regressively test the application during all stages and perform various simulations after final implementation of the project to ensure proper workflow of the system.

Replication:

In building our chat application, we will prioritize data redundancy and availability across multiple servers to safeguard against failures and ensure uninterrupted service during server or network issues. By implementing database replication, we can achieve high availability through configured replica sets for MongoDB or a multi-data center replication for Cassandra. Additionally, we'll replicate message queues, ensuring Redis replication for inter-server communication to maintain consistent queues. Our strategy also involves load balancing, managing application server replication behind a load balancer like Nginx or HAProxy.

Responsibilities

Client

- The client is responsible for interacting with end users, providing them a way to choose a person and send a message to them.
- React with Next.js will be used as the basis for this.

Server

- Socket io
- Next.js (node.js would probably be easier to use if none of us are familiar with next.js)
- Get started | Socket.IO
- Server will need to utilize the user mapping table to send a message to the correct Topic within the Redis Queue

Proxy

- Forward request to primary server: The proxy will receive incoming connections from clients and will forward these requests to the primary chat server or servers that are up and running.
- Load balance request among active replicas: HAProxy will distribute the incoming client requests across multiple chat server instances to ensure even load distribution and optimal resource utilization. It will use various algorithms like round-robin, least connections, or source-based to effectively manage the load.
- It will be very helpful in cases when the servers go down where it would reroute the traffic to other available servers to ensure continuous service
- Proxy can also store frequently accessed data and do the caching, helping reduce the load on the backend servers and improving response time
- It adds an extra layer of security as well by encrypting/decrypting secure traffic before passing it on to the servers

Communication models

Client/Proxy will use the request/reply model

- The client sends a message request to the proxy. The proxy then forwards the request to the chat server. Once the server processes the message, it sends a reply back through the proxy to the client. This can be implemented using HTTP/S protocols where the client initiates a connection and waits for the response.

Proxy/Server will use a message queue

- The proxy can implement a message queue (Redis) to handle asynchronous communication between the proxy and the chat servers. This decouples the proxy requests from the server processing time. It allows the proxy to continue receiving client requests without waiting for the chat servers to respond. The servers can pick up requests from the queue as they become available to process them.

Implementation Details

Client

- **System requirements**
    - Request connection with server and keep connection.
    - Store a unique user ID to communicate with.
    - Able to communicate with the load balance**r**
- **System inputs**
    - User requests
    - User messages
- **Workflow of data**
  Scenario: User wants to send a message to another user

- Websocket connection with server established.
- User inputs their ID and password
- On correct input, get access to their message inbox and an option to send a new message.
- Can input user ID to send message to, then input message.
- Send message, which then sends this message over the websocket connection.

System Guarantees:

- The client will keep trying to connect to the server until it establishes a connection. If a connection that has been established is not responding, the connection is broken if the request takes longer than a certain time period and the client tries to re-establish a connection with the server

Evaluation:

- The client would be set up as a react front-end that sends and receives responses from the server through the proxy.

Proxy

- **System Requirements:**
  - HAProxy installed and configured on a dedicated machine.
  - Ability to handle incoming client connections and distribute them among multiple backend servers.
  - Implement health checks to monitor server availability and reroute traffic in case of server failures.
  - Support for session persistence if required by the application.
  - Enable SSL termination for secure communication between clients and servers.
  - Utilize logging and monitoring tools for performance analysis and troubleshooting.
- **System Inputs**
  - Incoming client connections.
  - Requests from clients for message transmission.
  - Status updates from backend servers regarding availability and load.
- **Caching**
  - HAProxy itself is not a caching layer, but it can be configured to work with external caching solutions with in-memory data structures like Redis. For a chat application, caching can be used to store frequently accessed data like user session information, commonly accessed chat rooms, or static resources.
- **Load Balancing**
  - HAProxy excels at load balancing and can be configured to use several algorithms:
    - Round Robin: Requests are distributed across the servers sequentially.
    - Least Connection: Requests are sent to the server with the fewest active connections.
    - Source: Requests from the same source IP address will be sent to the same server.
- **Scaling**
  - Scaling with HAProxy can be done both vertically (adding more resources to existing servers) and horizontally (adding more server instances). HAProxy can be configured to automatically detect new server instances and add them to the load-balancing pool.

- **Security**
    - HAProxy provides several security features like SSL termination, which allows you to handle encrypted traffic effectively. It can also be configured to block IP addresses, limit the rate of requests to prevent DDoS attacks, and ensure that only authenticated users can access the chat servers.
- **Fault Tolerance**
    - Redundancy: HAProxy supports multiple backend servers, so in the case of a server failing, other servers can take over the load. This redundancy is a primary method of achieving fault tolerance.
    - Health Checks: HAProxy regularly performs health checks on the backend servers. If a server is found to be unresponsive or unhealthy, HAProxy stops sending traffic to it until it becomes healthy again.
    - Session Persistence: For applications that require session persistence, HAProxy can be configured with sticky sessions to ensure that a user's session remains active even if one server goes down.
    - Graceful Degradation: The system can be designed to degrade gracefully under failure conditions. For instance, if a backend server fails, HAProxy can be configured to serve a cached version of expected responses or provide a read-only service if the primary write server is down.
- **Single Point of Failure**

  While HAProxy itself can be a single point of failure, there are strategies to mitigate this:
    - HAProxy Pair: Set up a pair of HAProxy servers in an active-passive or active-active configuration using a high-availability solution like Keepalived or Pacemaker. This way, if the active HAProxy server fails, the passive one takes over, ensuring continuous availability.
    - DNS Load Balancing: Use DNS-level load balancing to distribute requests across multiple HAProxy instances, providing another layer of redundancy.
    - Clustering: In some environments, HAProxy can be clustered to work together, sharing state and session information to provide high availability.

  In case of a failure in the HAProxy server:
    - Failover: The backup HAProxy server takes over the IP address and continues serving client requests.
    - State Sharing: If configured, the backup will have the current state of the sessions and can continue where the failed instance left off.
    - Notification: Administrators are notified of the failover event for corrective action on the failed instance

System Guarantees:

- Load Balancing: HAProxy evenly distributes incoming client connections among available backend servers, optimizing resource utilization and preventing any single server from becoming overloaded.
- High Availability: HAProxy monitors the health of backend servers and redirects traffic away from failed or unhealthy servers, ensuring continuous service availability for clients.
- Session Persistence: HAProxy maintains session persistence if required, ensuring that subsequent requests from the same client are routed to the same backend server, which is essential for certain applications.
- SSL Termination: HAProxy handles SSL termination, decrypting encrypted traffic from clients and forwarding it in plain text to backend servers, ensuring secure communication without imposing additional overhead on backend servers.
- Logging and Monitoring: HAProxy provides logging and monitoring capabilities to track performance metrics, identify issues, and troubleshoot any problems that may arise.

Evaluation:

- Performance Testing: Measure response time, throughput, and system resource utilization under varying loads to evaluate the effectiveness of load balancing and server health checks.
- Failover Testing: Simulate server failures and observe HAProxy's ability to reroute traffic to healthy servers without interruption in service.
- Scalability Testing: Assess HAProxy's ability to scale with increasing numbers of clients and backend servers while maintaining performance and reliability.
- Security Audit: Conduct a security audit to ensure that SSL termination is properly configured and secure against potential threats.
- Logging and Monitoring Analysis: Review logs and monitoring data to identify any performance bottlenecks or issues that need to be addressed for optimization.

Server

- **System requirements**
    - Open and maintain websocket connection with the client
    - Read and write into the User-server mapping table

    - Communicating with other server through Redis Message Queue
    - Able to communicate with the load balance**r**
- **System inputs**
    - User connection requests
    - User messages
    - Incoming messages from message queue
    - User/Server mapping response
- **Workflow of data**

  Scenario: User wants to send a message to another user
    1. Establish web socket connection with client (Ex: user 5 connects to server 1)
    2. Write to user/server mapping table. If User 5 is a new user we create User 5 in the mapping table. If User 5 is a returning user we update that specific user in the mapping table. (Ex: Write User 5 -> Server 1)
    3. User now wants to send a message to the other user (Ex: User 5 wants to send a message to User 4)
    4. Look up the server that the user is connected to in user/server mapping table (Ex: Server 1 looks up what server User 4 is connected to. This is Server 3)
    5. Send a message to the corresponding topic of the target server. (Ex: Server 1 sends User 5's message to Topic 3 that User 4 is connected to)
    6. The server subscribed to that message queue will receive the message from its topic. (Ex: Server 3 receives message from its corresponding topic, Topic 3)
    7. After the server receives the message, it will send it to the user over the websocket connection. (Ex: Server 3 sends message to User 4)

System guarantees

- When a user joins for the first time, their user id is correctly mapped to the correct server in the database. When a returning user joins, their user id is correctly updated in the database mapping.
- Server will be reliable and successfully send the message to the correct topic
- When the target server receives a message from its respective topic, it will successfully send the message to the correct client.
- **Uptime:** We will aim for the highest uptime possible. In the case that there is downtime in any of the server nodes, we will attempt to recover as quickly as possible (i.e. spinning up another node, or restarting that server)

Evaluation

- We will monitor the uptime of the servers periodically to ensure that we maintain a high uptime.

- Whenever a server goes down or crashes unexpectedly, then we should find out why it went down and what were the conditions that caused the crash.
- We can evaluate the response times for messages to be sent from one user to another user. We can also do this between servers, where we measure the time it takes for a message to be sent from a source server to a target server. Using these response times, we can monitor if the system is slowing down for some reason (relative to internet/network speeds).
- Implement some sort of end-to-end testing to ensure that servers are reliable and functioning correctly.

Replication

Which process will be replicated?

- The server processes will be replicated.

What is the purpose of replication? (Availability? Fault Tolerance?)

- The purpose for replicating the servers is to have a system that is available in the case that a single server goes down. By having multiple servers that the front end can connect to, this avoids a single point of failure. If a server that a client is connected to goes down, then we can reconnect that client with another server to continue serving the clients requests.

Type of Replication? (Active? Passive?)

- Our architecture uses more of the concepts from active replication in the sense that every server replica processes requests in the same sequence. However, our architecture differs from active replication by not sending each client request to every server. Instead, we have a load balancer that distributes requests equally. Given the way we are using WebSockets to connect the client and the server, one client will connect with only one server, meaning that it is not useful to send requests to all servers.

How consistent should replicas be?

- In our architecture for the servers, we don't need to handle all the requests on each of the servers. Each server will handle the requests that are routed to them in a similar manner. The servers all communicate to the same database for the user/server mapping table to remain consistent. Communication between the servers is handled by the Redis message queue.

Group Communication Implementation?

- We will use a Redis message queue to communicate between servers. When a server needs to send a message to another server, it can send the message into that server's associated topic. When a server has a message in its associated topic, it can retrieve that message from its topic. Any given server can find out which server a user is connected to using the user/server mapping table in the database.

Group Management needs?

- To manage which requests go to which server, we are going to use a proxy (load balancer) in order to route requests to the proper server, ensuring that one server is not overloaded by requests.
- In the case that a server crashes, the rest of the group does not need to know that a replica has crashed. Instead we will ensure that every server is responsible for its own crashes, and can successfully restart its server process whenever it does crash. This decouples each server from the rest of the replicas by making each server responsible for its own crashes without affecting the rest of the group.

Fault Tolerance

Client:

How does the Client respond when a process fails? (server, proxy)

- If the server that the client is connected to crashes, then the proxy will reroute the client to another server in the distributed system.
- If the proxy fails, then the client will retry the connection to the proxy after a certain amount of time has passed through timeouts.
- Omission failure: When the client sends a message, it will wait for an acknowledgement from the server. If it does not get the acknowledgement back within a certain time (5 seconds) after sending the message to the server, it will resend the message to the server side.

How will the Client recover if it goes down?

- If the client itself crashes then it will attempt to restart itself until it successfully manages to do so.

Can requests be executed more than once?

- Yes, when the server is sending messages to a specific client. If the server did not receive an acknowledgement within a set time frame from the client, then the server will resend the message to the client. All messages have an ID attached to them which is generated for a unique message that is sent. If it is a retry, then the ID is the same for all retried messages to the original. This way, if a message was received, then the ID is checked if it is greater than the previous IDs to allow the message to be added.

Proxy:

HAProxy helps in achieving fault tolerance by managing traffic and checking the health of the servers. If one server stops working, HAProxy quickly redirects traffic to other working servers without users noticing. This helps to recover the issues when a single point of failure occurs and makes sure that the application keeps running smoothly.
We are implementing Keepalived Mechanism to set up an active-passive implementation for the proxy , to make sure we have a reliable backup ready to ensure there is no interruption. This mechanism will help the most when the active proxy goes down, and then a backup proxy will help run things. Keepalived uses the “heartbeat” mechanism to

constantly check if the main server is alive and can take immediate action if it is not. This setup significantly reduces the chances of any downtime, maintaining constant availability. Load balancing through proxy plays a major role to prevent overloading and reduce the chances of failures.

Redundancy with HAProxy

- Deploying multiple instances across different servers and configuring HAProxy to distribute client requests across these servers help deter redundancy
- If one server fails, HAProxy detects this and automatically reroutes traffic to the remaining healthy servers, and ensures failure of single server does not affect the availability of the system

Health Checks

- HAProxy is configured to perform regular health checks on the backend servers at regular intervals and require servers to respond multiple times to ensure the healthy behavior of the servers
- If a health check fails, HAProxy simply removes the server from the active pool of servers and shifts to the backup servers and constantly checks the health of these backup servers as well, and moves from one backend server to another until a healthy server is found.
- The unresponsive server is added back to the pool once it passes the health checks

Session Persistence

- Configuring HAProxy with “sticky sessions” ensures that a client is consistently connected to the same backend server for the duration of their session and is achieved through the “stick-table” implementation in the proxy
- This is really helpful for maintaining consistency of a user’s interaction in the application

Graceful Degradation

- It is also possible for HAProxy to serve cached-responses if all of the backend servers are down, ensuring that users still receive data.
- This can portray as a “read-only” effect for the users till the time servers are up and running

Failure Mechanisms & Single Point of Failure

- Using Keepalived to develop an active-passive model for the proxy helps prevent major faults such as when Proxy fails. The passive proxy takes over the system when the active proxy fails
- Also, the backup instance can pick up where the current session was left off and continue from the same state, ensuring uninterrupted service

DNS Load Balancing

- DNS balancing across multiple load balancers combined with DNS health checks automatically routes traffic away from failed load balancers. This overcomes the faults when load balancers are not in effect in the proxy

Queueing of Requests

- In case of spikes in traffic beyond control, HAProxy can use its own queue for queueing messages and can use timeouts in the queue it creates to drop the request if it is timed-out, and can help prioritize the messages to be delivered based on some attributes. We will implement this feature if time persists (because we are already making use of the Redis queue).

We can also setup a notification system for the administrators to diagnose and identify the faults as soon as they occur so that right action can be taken before it is too late

How does Proxy respond if Server and Client fail?

If a server fails, the proxy (such as HAProxy) will detect this failure usually through health checks or failed connection attempts. Once the proxy detects that the server is unresponsive, it will reroute the traffic to other healthy servers in the pool. This redirection is often seamless and automatic, preventing users from experiencing any disruption in service.

For client failures, the proxy's role is a bit different. A proxy doesn't usually respond directly to client failures because clients are typically the initiating party of a request. If a client fails after making a request:

- The proxy will attempt to send the request to the server.
- If the client cannot acknowledge the response because it has failed (crashed or disconnected), the proxy will eventually close the connection due to a timeout since it won't receive an acknowledgment from the client.

In an active-passive HAProxy setup with Keepalived:

- If HAProxy itself fails, Keepalived will detect the lack of a "heartbeat" (a signal indicating that the proxy is alive) from the active proxy.
- Keepalived will then switch the virtual IP (VIP) to the passive HAProxy server, making it the new active server.
- The new active HAProxy server will start handling requests from clients and routing them to the available servers.

HAProxy and Keepalived together ensure that the failure of servers or the proxy itself has a minimal impact on the availability and reliability of the application from the client's perspective. However, the proxy at maximum manages client connection failures through timeouts.

Crash Failure

- If a server crashes and stops working, HAProxy detects this and stops sending it new users, automatically rerouting them to healthy servers.

Omission Failure

- HAProxy helps to retry sending messages to detect and recover from omission failure

Byzantine Failure

- HAProxy cannot handle these failures alone, however can help by routing requests across the backend

Server:

How does System respond when process fails? (Crash failure? Omission failure? Byzantine failure?)

- The client connected to the server which fails will be reconnected to another server. The failing server will recover by restarting it self.
- Omission failure: We will send an acknowledgment back to the client when the server receives a message from the client. This will ensure that the client is aware if the message reached the server. If the client doesn't receive the acknowledgment, then after 5 seconds, the client will be notified that the message failed to send, and will be allowed to retry.

How will the system recover?

- In the case that a server crashes or unexpectedly goes down, the server will successfully restart its server process.
- We will use a tool called PM2, which is a Node.js process manager that allows certain actions to be automatically programmed to occur whenever a process fails or crashes. This will allow servers to be restarted automatically when they crash.

Can request be executed more than once?

- When the client sends a message to the server, if the acknowledgment fails to make it to the client from the server, then the client will have the chance to resend the message. However, in this case, if the initial message was still successfully received and processed by the server, then we will have duplicated messages after the retry is attempted. To remedy this, we will implement a check whenever the server receives a retried message front the client it will check in the database to see if the message was previously processed or not.

Group management?

- In the case that a server crashes, the rest of the group does not need to know that a replica has crashed. Instead we will ensure that every server is responsible for its own crashes, and can successfully restart its server process whenever it does crash. This decouples each server from the rest of the replicas by making each server responsible for its own crashes without affecting the rest of the group.
- The proxy is in charge of reconnecting the clients with another server if a server ever goes down. The proxy will also be checking the servers regularly to ensure that every server is able to respond appropriately.

—

Case 1: Sending a message from client 1 to client 2.

If client 1 wanted to client 2, show user server mapping in db to check.

DB - user server mapping, client message IDs.

Synchronization

We have decided to pivot away from HAProxy to implement a form of Leader Election among the servers to elect a leader server that will act as the load balancer. We will use either the Ring Algorithm or Bully Algorithm to elect a leader server.

Leader Server:
The leader server will assume the role of the proxy, distributing incoming requests via round robin algorithm. If the leader server fails, a new leader server will be elected among the follower servers using either Ring Algorithm or Bully Algorithm. This newly elected follower server will take on the responsibilities of proxy.

Follower Server:
The follower server will process requests that it receives from the leader server (load balancer). It will perform the same functions that were previously mentioned for communication, including sending/receiving messages to/from the Redis message queue.

Concerns for consistency:

Message IDs

- We can have clients sending a unique message id with every message they send to the server.
- The server will send an Ack back to the client when they receive a message successfully
- Client will receive the Ack and be sure that the message is sent. If the client does not receive an Ack within a certain timeframe, it retries sending message again
- When the server receives a message again, it checks for the id of the message in its database/queue to check if that message id exists or not.
- If the message id exists, the server ignores the message and sends an Ack to the client, and the client does not retry sending messages.
- If the message id does not exist, server accepts the message and forwards it to the message queue and send Ack back to the client

Causal Consistency

Causal Consistency helps ensure that users have a consistent view of data. Maintaining a proper order of messages and updates will enhance user experience. MongoDB helps in achieving data consistency through replication. It uses replica sets to replicate data

throughout multiple servers. We can have a set number of replicas approving the write operations and read operations before it is considered successful.