Implementing a Pub-Sub Architecture for Real-Time Organizational Notifications

Authors(alphabetically): Anwitha Arbi[#1], Rebecca Dsouza[#2], Simran Dhawan[#3]

[#] *Computer Science Department, Santa Clara University*
Santa Clara, CA 95053, USA

[1] aarbi@scu.edu / *SCU ID- 07700005478*
[2] rcdsouza@scu.edu / *SCU ID- 07700005837*
[3] sdhawan@scu.edu / *SCU ID- 07700011429*

***Abstract*— This paper presents a live news notification system designed to provide real-time updates on various topics such as finance, hiring, and more within an organization. The system is built on the publisher-subscriber model of message delivery, ensuring timely and relevant information dissemination. Key features of the system include fault tolerance, scalability, data integrity, consistency, security, concurrency, and performance. To achieve these objectives, the system employs several algorithms and protocols, including the Heartbeat Protocol for monitoring server status, a Replication and Consistency Protocol to ensure data integrity and consistency across distributed systems, a Concurrency Protocol to handle simultaneous data accesses, and a Broadcast Protocol for identifying presence of broker nodes. The system's architecture utilizes dual brokers with a primary-secondary setup, hosted on separate Amazon EC2 instances, to enhance reliability and fault tolerance. This paper details the system's design, implementation, and the results of various tests conducted to evaluate its performance and effectiveness.**

***Keywords*— Pub-Sub, Flask API, Multi-threading, Heartbeat Protocol, Multi Broker**

I. Introduction

In today's organizational environments, the demand for real-time, reliable information dissemination is paramount. This paper introduces a live news notification system built on a robust publisher-subscriber model, designed to meet this demand. Utilizing a fault-tolerant, distributed architecture with a dual-broker setup, the system ensures continuous information flow even in the event of failures. Key to its design is the integration of multi-threading for efficient concurrency control and the Flask framework for backend flexibility. This system not only guarantees timely delivery of organizational news but also emphasizes security and data integrity, making it an essential tool for modern corporate communication.

II. Project Goals

A. At Least Once Message Delivery:

Objective: Subscribers of the topic should be able to receive messages at least once.

Real-World Impact: We have created a fault tolerant Broker based publisher-subscriber system which will be able to provide at least once message corresponding to subscribed topics even when the client is offline or one of the brokers fails.

B. Enhanced Real-Time Communication:

Objective: To minimize the delay in disseminating critical information within an organization.

Real-World Impact: By providing instant updates on a variety of topics, the system aims to streamline internal communication, ensuring that all members of an organization are promptly and uniformly informed. This is particularly crucial in scenarios requiring quick decision-making or immediate action.

C. Efficient Handling of Consistency :

Objective: Consistency between two brokers should be maintained

Real-World Impact: This goal addresses the challenge of maintaining system responsiveness, especially during crashes in the system, i.e when brokers are joining or leaving the system. By effectively handling high replication and consistency, the system ensures that users receive timely updates and that their interactions with the system are smooth and uninterrupted.

D. *System Reliability and Availability:*

Objective: To create a system that remains operational and accessible even during partial failures or maintenance activities.

Real-World Impact: The goal is to tackle common issues of system downtime in organizational settings. By ensuring that the system is fault-tolerant, organizations can rely on uninterrupted communication for critical operations, thereby reducing the risk of operational disruptions and enhancing overall productivity.

E. *Adaptability to Organizational Growth:*

Objective: To design a system that can scale efficiently in response to an increasing number of users and growing data demands.

Real-World Impact: The brokers of the system identify each other's presence via broadcast protocol. Also the system is made menu driven where new brokers can join or leave the system anytime they want.

F. *Secure and Controlled Information Flow:*

Objective: To safeguard sensitive information and ensure that it is accessible only to authorized individuals.

Real-World Impact: In an era where data breaches and unauthorized information access are major concerns, the system's focus on security is vital. By controlling who receives notifications and ensuring data integrity, the system helps maintain confidentiality and trust within the organization.

III. Motivation

Our project was motivated by the noticeable absence of systems that enable users within an organization to subscribe selectively to specific topics of interest for receiving live updates. Recognizing this gap, we identified the need for a robust messaging system capable of withstanding failures without disrupting service. This necessity is further driven by the critical importance of ensuring message reliability and redundancy. Our aim was to develop a solution that not only addresses these specific needs but also enhances the overall efficiency and effectiveness of communication in organizational settings.

IV. Distributed System challenges addressed

In developing our live news notification system, we confronted and addressed several fundamental challenges associated with distributed systems. A paramount concern was ensuring fault tolerance. Our approach involved the implementation of a dual-broker setup, which guaranteed continuous service by allowing one broker to seamlessly take over in the event of a failure in the other. This design was crucial for maintaining uninterrupted operations.

Replication was performed to handle fault tolerance but data consistency presented significant challenges in environments where real-time updates were crucial. Our system addressed this by replicating metadata at both primary and secondary broker queues. This ensured consistent data availability, even during broker transitions, thereby maintaining the integrity of the communication process.

Scalability is another critical aspect of distributed systems. Also brokers are added and removed dynamically using POST requests that make the broker scalable. Our system leveraged Amazon EC2 instances, providing the flexibility to scale resources in response to varying loads, whether due to an increase in user numbers or message volume. This scalability was essential for accommodating organizational growth and fluctuating demands.

Concurrency control was also a key challenge in distributed systems, particularly when ensuring system performance and responsiveness. Our solution employs multi-threading to effectively manage concurrent requests, thus preserving system efficiency and user experience. Heartbeat algorithm which continuously runs was presented on one separate thread to prevent blocking.

Security in distributed systems, especially in terms of data access and message delivery, is of utmost importance. Our system enhances security by restricting message receipt to registered subscribers only, thereby safeguarding against unauthorized access and ensuring data confidentiality.

Furthermore, network latency and overall performance were critical considerations in real-time notification systems. Our design focused on minimizing latency and optimizing performance to ensure the timely delivery of notifications, a crucial feature for organizational communication.

Lastly, the complexity and maintenance of distributed systems could be daunting. We addressed this by developing a user-friendly, menu-driven program for the straightforward setup and configuration of brokers, publishers, and subscribers, thereby reducing the system's operational complexity and easing maintenance requirements.

By tackling these challenges, our system not only demonstrates a comprehensive solution to the intricacies of distributed systems but also significantly enhances the efficacy of communication within organizational settings.

V. PREVIOUS WORK

In the field of edge computing and IoT communication protocols, MQTT has been a focus of significant research due to its efficient data handling in IoT applications. Donta and Dustdar (2023) emphasized MQTT's robustness in IoT, highlighting its utility in edge computing environments [1]. Milica Matic and colleagues (2020) extended this work by optimizing MQTT message routing in cloud architectures, targeting improvements in load distribution and system scalability [2].

Longo, Redondi, Cesana, Arcia-Moret, and Manzoni proposed MQTT-ST, a protocol inspired by the Spanning Tree Protocol (STP), aiming to overcome the limitations of centralized MQTT brokers [3]. They developed MQTT-ST by modifying the MQTT protocol to include mechanisms for broker interconnection, path computation based on latency and broker resources, and a dynamic selection of root brokers. This approach led to increased fault tolerance and flexibility in distributed networks. MQTT-ST's effectiveness was validated through stress tests in simulated environments, demonstrating improved publication throughput and reduced end-to-end delay compared to traditional centralized MQTT brokers [12-21].

In parallel, Kafka, introduced by J. Kreps (2011), has gained attention as a distributed messaging system [4]. It employs a "pull" model to efficiently handle message traffic, reducing message loss and flooding, thereby enhancing system scalability and efficiency. This model is especially beneficial in distributed publish-subscribe systems.

Another notable advancement in this field is the introduction of "PopSub" by Salehi, Zhang, and Jacobsen [5]. PopSub addresses the inefficiencies in resource utilization in distributed content-based publish/subscribe systems. It employs a popularity-based publication routing algorithm, which takes into account both broker resources and the popularity of publications among subscribers. By measuring the impact of forwarding a publication on the overall delivery of the system, PopSub significantly improves broker resource efficiency (by up to 62%) and reduces delivery latency (by up to 57%) under high load conditions. This approach contrasts with existing methods that often involve costly overlay reconfigurations or are optimized only for large clusters of similar subscriptions without considering isolated ones. PopSub's methods include Direct Delivery, Batching, and Gossiping, offering scalable alternatives for handling unpopular publications [25-32].

Ghosh’s work on "Message Diffusion" in distributed publish-subscribe systems presents a novel method for efficient information distribution within embedded and IoT networks. This approach utilizes a publish and subscribe model, involving event publishers, services, and subscribers. It emphasizes the importance of decoupling events in terms of time, space, and synchronization to enhance scalability and adaptability in distributed environments. By applying filters for selective forwarding and content-based routing, this model efficiently matches events to specific subscribers [6].

VI. Project Design

A. Key Design Goals

1) Heterogeneity

We have Flask Libraries to set up servers and Amazon EC2 instances for hosting each broker, the system is made adaptable to various operating environments, ensuring seamless operation regardless of the underlying hardware or software.

2) Openness

The system's use of a menu-driven program for setting up brokers, publishers, and subscribers allows for straightforward integration and scalability. This approach ensures that the system can be easily adapted or expanded to meet evolving organizational needs. The brokers can join or leave the system without impacting the publishers and subscribers.

3) Security

Security is enforced by allowing only registered subscribers to receive messages. This approach ensures that sensitive information is not exposed to unauthorized entities, maintaining the confidentiality and integrity of organizational communications.

4) Failure Handling

The Heartbeat protocol between the two brokers (Broker 1 and Broker 2) monitors the status of each broker. If the primary broker of a pair goes down, the secondary broker takes over, ensuring that there is no disruption in message delivery to subscribers.

5) Concurrency

The system uses multi-threading to handle concurrent requests and operations. This is particularly important in managing the simultaneous activities of publishers posting messages and subscribers receiving updates.

6) Quality of Service

At least once message delivery is ensured. We have created a fault tolerant Broker based publisher-subscriber system which will be able to provide at least once message corresponding to subscribed topics even when the client is offline or one of the brokers fails.

7) Scalability

The deployment on Amazon EC2 instances along with usage of POST api of flask web server to dynamically join or exit the system provides the necessary infrastructure to scale resources as required. This scalability is crucial for accommodating an increasing number of publishers and subscribers without degrading system performance.

8) Transparency

Despite the underlying complexity of the distributed system, the user interface is kept straightforward, particularly through the flask web server commands particularly GET and POST which are accessed via menu-driven program. This approach allows users to easily interact with the system, whether setting up brokers or subscribing to topics, without needing to understand the technical details of the distributed operations.

B. Key Components

1) Brokers:
Function: Act as intermediaries between publishers and subscribers.
Details: There are two brokers in the system, Broker 1 and Broker 2. One broker acts as the primary broker and other as the secondary (replica) broker. This dual-broker setup enhances fault tolerance and ensures continuous message delivery. Each broker is hosted on a separate Amazon EC2 instance.

2) Message Queues:
Function: Two types of queues
a) One Per Topic and corresponding data published by publishers.
b) One Per Subscriber per topic and index up to which topic is delivered.
Details: Each broker maintains its own message queues corresponding to a topic. Primary broker messages the secondary broker to replicate metadata and state information at secondary broker. This ensures message availability, consistency and fault tolerance.

3) Ops_ file

Function: Helps in restoring the state of system when any broker crashes at restarts.
Details: Whenever any broker goes down and it restarts its state is automatically resumed by metadata information present in the ops file. Each broker makes its own ops file and uses it to restart. This prevents loss of previous data even when both the brokers crash.

4) Publishers:
Function: Send messages to the system.
Details: Publishers are entities (or users) that send messages to the system. They specify a topic and a message using POST APIs on Flask webserver, which is then queued in the broker for delivery to subscribers.

5) Subscribers:
Function: Receive messages from the system.
Details: Subscribers are entities (or users) that sign up to receive updates on specific topics. They receive messages from the primary broker, or the secondary broker if the primary is unavailable.

6) Menu-Driven Program:
Function: Facilitate system setup and interaction.
Details: This program provides a user interface for setting up and managing brokers, publishers, and subscribers. It simplifies the process of configuring the system and managing subscriptions.

B) Algorithms

Heartbeat Protocol:
The heartbeat protocol is implemented between two brokers. Each of the brokers send the heartbeat to all the replicas with information as its own ip. This way each broker tracks which replica gets information when any broker goes down. The heartbeat is sent every 5 seconds, so failure in the system is detected after 5 seconds.

Performance of Algorithm:
Bandwidth: Every 5 seconds 1 message per broker is sent and n messages are received corresponding to n replicas. Here replicas are 1. So 1 message is received and 1 message is sent by each broker.

Replication and Consistency Protocol:
To maintain the consistency between primary broker and secondary broker. Primary broker receives a request from the client and the primary broker sends the same request to the replica if it is alive. If it is not alive the request towards the replica will not succeed and it will store the request as a pending request to be sent to the replica as soon as it comes up(It will receive information about the replica coming up from the heartbeat protocol implemented). Only after the state in the secondary broker(if alive) is updated along with the primary broker, will it send the response towards the client. Also, whenever any broker receives the message it stores that message in an ops file which is used by the broker to recover its state once it goes down.

Thus protocol implemented is fault tolerant and states at two brokers can recover as soon as they come up. Since it is two broker systems, failure of any one system will not impact the performance.

Performance of Algorithm:
Bandwidth: 4N messages in success scenario and 6N messages in failure scenario. So Complexity is O(N) messages, where N is the number of requests coming from publisher/subscriber.
Reliability: Prevents message loss and ensures at least once message delivery system.

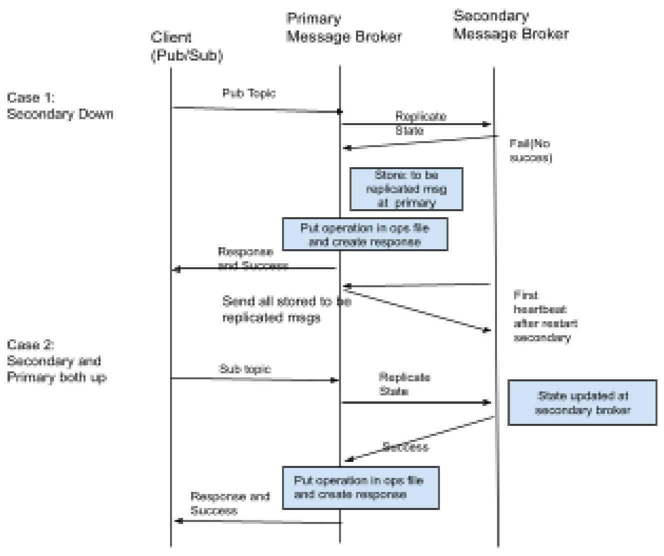

Client (Pub/Sub)
Primary Message Broker
Secondary Message Broker
Case 1: Secondary Down
Pub Topic
Replicate State
Fail(No success)
Store to be replicated msg at primary
Put operation in ops file and create response
Response and Success
Send all stored to be replicated msgs
First heartbeat after restart secondary
Case 2: Secondary and Primary both up
Sub topic
Replicate State
State updated at secondary broker
Success
Put operation in ops file and create response
Response and Success

Fig. 1 Replication and Consistency Protocol

Concurrency Protocol:

Message brokers concurrently send messages to multiple subscribers. We use Optimistic concurrency control in our system because we have a valid assumption that each object can be touched by one subscriber/ publisher for subscribing or publishing topics. Message broker is creating separate queues for publisher topic and data and separate queue for each subscriber and its topic and data index. So, chances of race conditions are unlikely here. If some issue happens we have stored operation files we can always rollback and restart the system.

Performance of Algorithm:

Bandwidth: No wait time for any publisher or subscriber so message exchange is with minimum delay. However if rollback when something fails then 2N messages are exchanged where N is the number of messages exchanged before the system detects requirement of rollback.

Broadcast Protocol

Two message brokers identify the presence of each other via broadcast protocol. The two message brokers add each other as a list of replicas from the menu. In future, when more than two brokers are there then their presence can also be known.

Performance of Algorithm:

Bandwidth: n^2 messages are exchanged corresponding to n brokers. Here brokers are 2. So identifying the presence of each broker via broadcast algorithm requires 4 message exchanges.

C. *Architecture*

The distributed system architecture depicted in the diagram (Fig. 2) operates on a publish-subscribe model, enhanced with mechanisms for reliability and fault tolerance. It consists of publishers, brokers, and subscribers, utilizing EC2 instances for the broker servers.

Publishers: There are multiple publishers responsible for generating and sending messages or data to a topic.

Brokers: The system includes two EC2 instances acting as brokers, with Broker 1 as the primary and Broker 2 as the secondary. These brokers are responsible for managing the distribution of messages from publishers to subscribers. The presence of a secondary broker indicates a redundancy plan to handle failover.

Operational (Op) File: Each broker maintains an operational file where it logs messages received. This file is crucial for the broker to recover its state in case of a failure, ensuring that no data is lost and the system can resume operation seamlessly after a disruption.

Subscribers: Multiple subscribers listen for messages on specific topics they are interested in. The system is designed to handle multiple subscribers, which can scale as needed.

Terminal Interface: This interface is presumably used for setting up and configuring the broker servers.

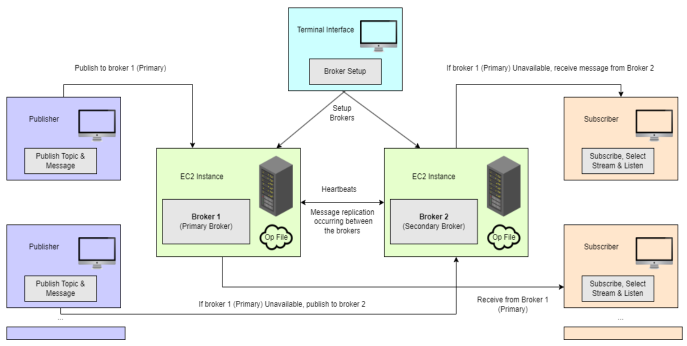

Terminal Interface
Broker Setup
Setup Brokers
Publish to broker 1 (Primary)
If broker 1 (Primary) Unavailable, receive message from Broker 2
Publisher
Publish Topic & Message
EC2 Instance
Broker 1 (Primary Broker)
Op File
Heartbeats
Message replication occurring between the brokers
EC2 Instance
Broker 2 (Secondary Broker)
Op File
Subscriber
Subscribe, Select Stream & Listen
Publisher
Publish Topic & Message
Subscriber
Subscribe, Select Stream & Listen
If broker 1 (Primary) Unavailable, publish to broker 2
Receive from Broker 1 (Primary)
...
...

Fig. 2 System Architecture

1) *Use Case Diagram*

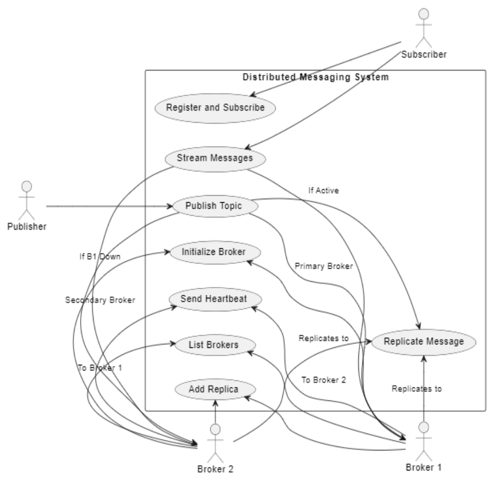


Fig. 3 Use Case Diagram

The Use Case Diagram presents a high-level overview of the distributed messaging system's functionality. It identifies the system's primary actors, the Publisher and the Subscriber, and their points of interaction with the system, such as registering, subscribing, publishing topics, and streaming messages. The diagram also outlines the system's ability to handle broker initialization and failover processes, showcasing how the system maintains its operations when a primary broker is down by utilizing a secondary broker.

2) *Sequence Diagram*

The Sequence Diagram provides a detailed chronological representation of interactions within the distributed messaging system. It specifies the precise order of messages exchanged between different instances, such as two broker instances and client applications (publishers and subscribers). The diagram traces the steps from the initial setup of brokers to the dynamic process of handling message publications and subscriptions, including the replication of data and heartbeats signaling to ensure system reliability and consistency.

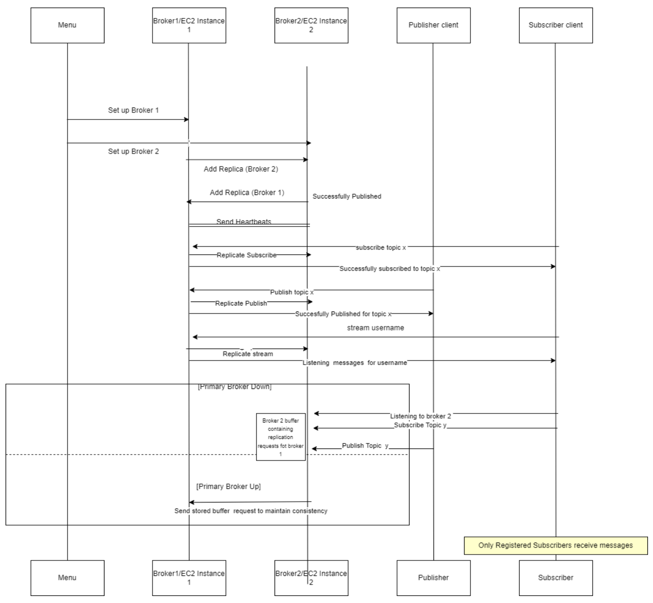


Fig. 4 Sequence Diagram

3) *Activity Diagram*

The Activity Diagram depicts the procedural flow and decision-making processes within the distributed messaging system, beginning with the setup of broker instances via a terminal. It maps out the steps for adding replica brokers, initializing them, and regularly sending heartbeats to maintain system awareness. The diagram also incorporates the system's response to the primary broker's status changes, illustrating the contingency actions taken when the primary broker is down, ensuring continuous message flow through the secondary broker.

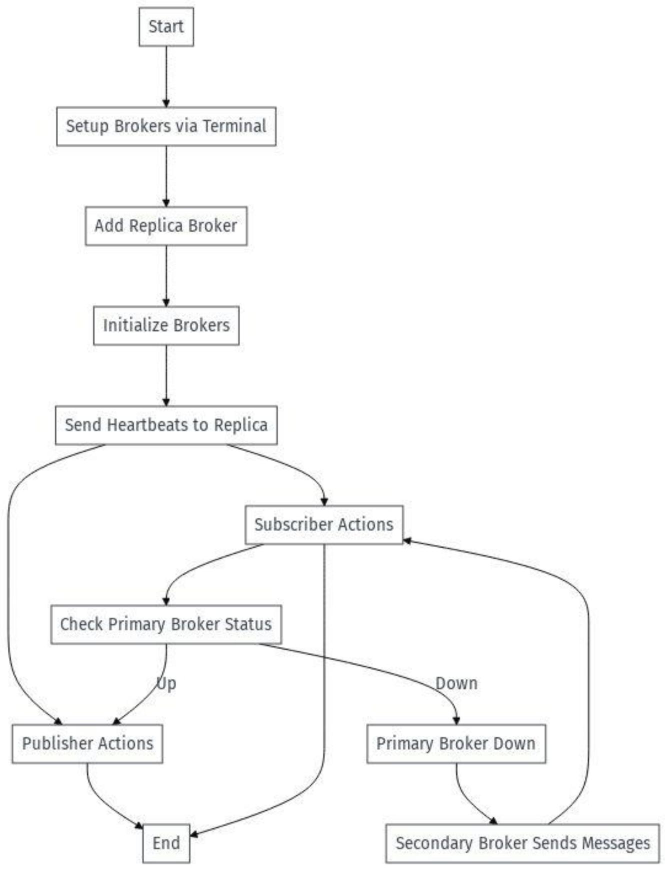

Start
Setup Brokers via Terminal
Add Replica Broker
Initialize Brokers
Send Heartbeats to Replica
Subscriber Actions
Check Primary Broker Status
Up
Down
Publisher Actions
Primary Broker Down
End
Secondary Broker Sends Messages

Fig. 5 Activity diagram

4) Class Diagram

Class Diagram presents the foundational structure of your distributed system. It includes four main classes: Broker, Publisher, Subscriber, and MenuDrivenProgram. The Broker class is responsible for initializing, sending heartbeats,

receiving, and replicating messages. The Publisher class manages the publishing of topics and sending messages. The Subscriber class handles registration, subscription, and streaming of messages. Lastly, the MenuDrivenProgram class is utilized for setting up brokers, publishers, and subscribers. This diagram is crucial for understanding the roles and responsibilities of each class in the system and their interactions.
aa

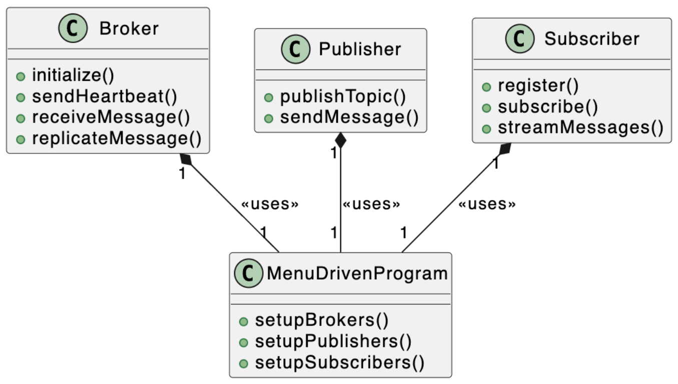


Fig. 6 Activity diagram

Class Diagram presents the foundational structure of your distributed system. It includes four main classes: Broker, Publisher, Subscriber, and MenuDrivenProgram. The Broker class is responsible for initializing, sending heartbeats, receiving, and replicating messages. The Publisher class manages the publishing of topics and sending messages. The Subscriber class handles registration, subscription, and streaming of messages. Lastly, the MenuDrivenProgram class is utilized for setting up brokers, publishers, and subscribers. This diagram is crucial for understanding the roles and responsibilities of each class in the system and their interactions.

5) Object Diagram

Object Diagram provides a snapshot of the system at a particular moment, showing the state of objects and their relationships. It features instances of Broker1 and Broker2, each with defined roles as primary and secondary brokers. The Publisher object is shown with a specific topic and message, and the Subscriber object is depicted as registered. The diagram illustrates the replication of messages between the brokers and the flow of messages from the publisher to the broker and then to the subscriber. This diagram is instrumental in visualizing the real-time state and interaction of various components in your system.

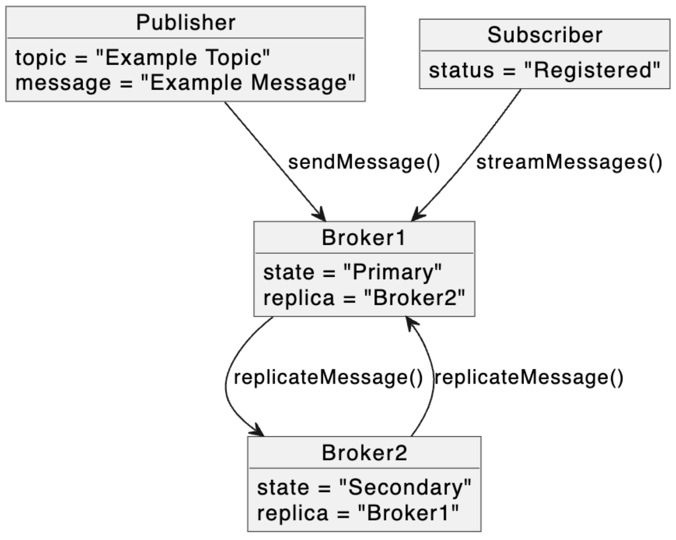


Fig.7 Object Diagram

6) Interaction Diagram

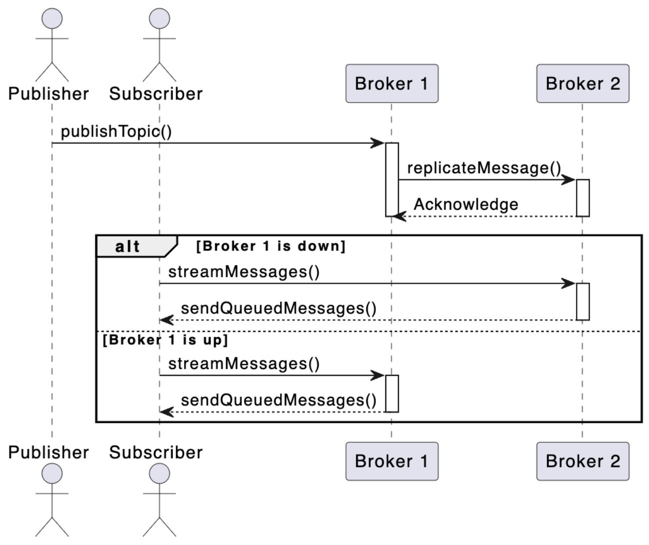
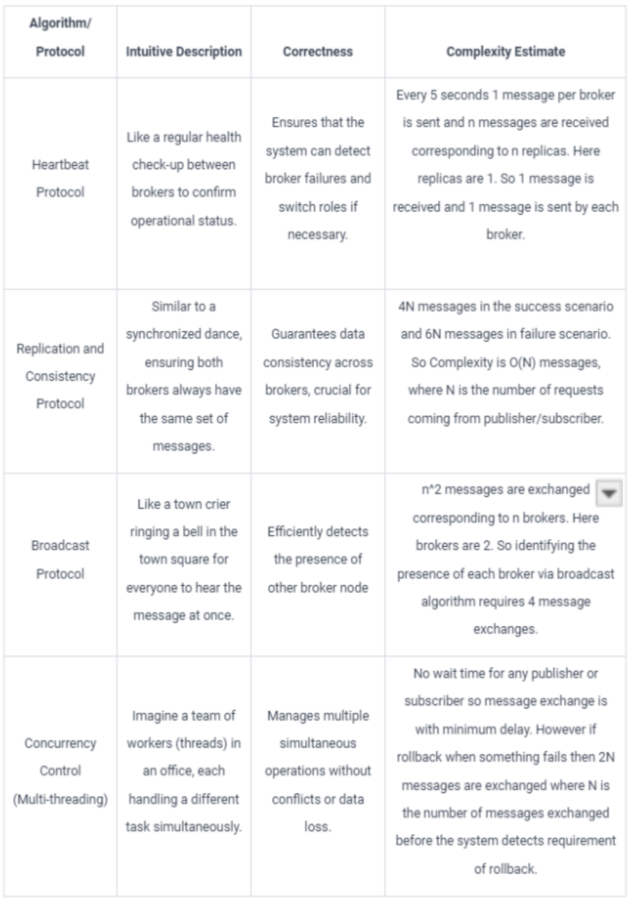


Fig.8 Interaction Diagram

Interaction Diagram focuses on the sequence of interactions between Publishers, Subscribers, and Brokers. It shows how a Publisher publishes a topic to Broker 1, which then replicates the message to Broker 2. In scenarios where Broker 1 is down, the Subscriber streams messages from Broker 2; otherwise, it streams from Broker 1. This diagram is key to understanding the dynamic behavior of the system, especially how it handles failover scenarios and ensures continuous message streaming.

VII. Evaluation



Algorithm/ Protocol	Intuitive Description	Correctness	Complexity Estimate
Heartbeat Protocol	Like a regular health check-up between brokers to confirm operational status.	Ensures that the system can detect broker failures and switch roles if necessary.	Every 5 seconds 1 message per broker is sent and n messages are received corresponding to n replicas. Here replicas are 1. So 1 message is received and 1 message is sent by each broker.
Replication and Consistency Protocol	Similar to a synchronized dance, ensuring both brokers always have the same set of messages.	Guarantees data consistency across brokers, crucial for system reliability.	4N messages in the success scenario and 6N messages in failure scenario. So Complexity is O(N) messages, where N is the number of requests coming from publisher/subscriber.
Broadcast Protocol	Like a town crier ringing a bell in the town square for everyone to hear the message at once.	Efficiently detects the presence of other broker node	n^2 messages are exchanged corresponding to n brokers. Here brokers are 2. So identifying the presence of each broker via broadcast algorithm requires 4 message exchanges.
Concurrency Control (Multi-threading)	Imagine a team of workers (threads) in an office, each handling a different task simultaneously.	Manages multiple simultaneous operations without conflicts or data loss.	No wait time for any publisher or subscriber so message exchange is with minimum delay. However if rollback when something fails then 2N messages are exchanged where N is the number of messages exchanged before the system detects requirement of rollback.

Table 1. Providing intuitive description of the algorithms, their correctness and their complexity

Flask Web Framework:

Advantages: Flask is lightweight and highly scalable, making it ideal for handling HTTP requests in a distributed system. Its simplicity and extensibility allow for rapid development and easy integration with other components like message queues.

Why It's Good: Flask's flexibility is crucial for a system that requires adaptability and scalability, especially when dealing with real-time data communication in an organizational context.

Amazon EC2 Instances:

Advantages: EC2 provides scalable and reliable cloud computing resources. It allows the system to dynamically adjust computational resources based on demand, ensuring high availability and performance.

Why It's Good: The use of EC2 instances ensures that the system can handle varying loads efficiently, making it robust against traffic spikes and scalable as the user base grows.

Message Queues:

Advantages: They decouple the process of sending and receiving messages, allowing for asynchronous communication. This means publishers and subscribers don't need to interact with the system at the same time.

Why It's Good: This approach enhances the system's efficiency and reliability, ensuring that messages are not lost even if a subscriber is temporarily unavailable.

Heartbeat Protocol:

Advantages: Regularly checks the health of brokers, enabling quick detection and recovery from failures.

Why It's Good: This protocol is essential for maintaining high availability and fault tolerance in the system, ensuring continuous operation even in the event of broker failures.

Replication and Consistency Protocol:

Advantages: Ensures that all copies of the data across the system are consistent, which is crucial for data integrity in a distributed environment.

Why It's Good: This protocol guarantees that subscribers receive up-to-date and accurate information, which is vital for decision-making processes within organizations.

Broadcast Protocol:

Advantages: Efficiently helpful in identifying presence of other broker nodes in the system .

Why It's Good: The ability to broadcast messages quickly and reliably is helpful in joining the system by any new message broker.

Concurrency Protocol:

Advantages: Efficiently helpful maintaining consistency .

Why It's Good: Optimistic concurrency protocol is implemented because of the valid assumption that the system will have less conflicts because of separate queues implementation.

VIII. Demonstration

System Scenario I:

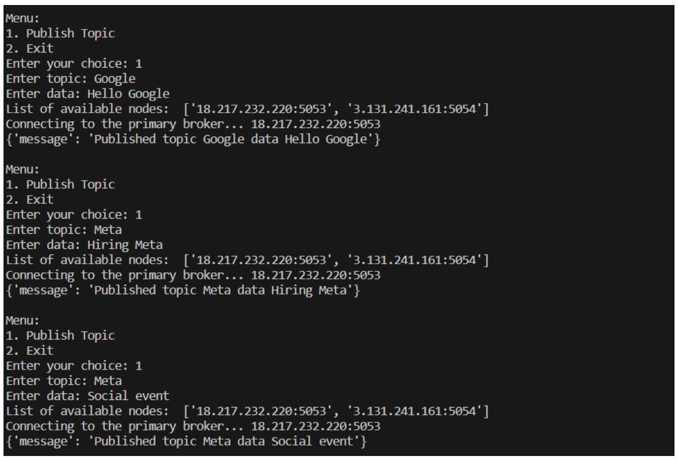
- Both primary and Secondary brokers are alive.
- Subscribers:
  - Username: Alice
    - topics: Google
  - Username: Bob
    - topics: Google, Meta
- Publishers:
  - Topic: Google
    - Hello Google
  - Topic: Meta
    - Hiring Meta
    - Social event

Results:

- Alice Listening messages(GET stream): Hello Google
- Bob Listening messages(GET stream): Hello Google, Hiring Meta, Social Event

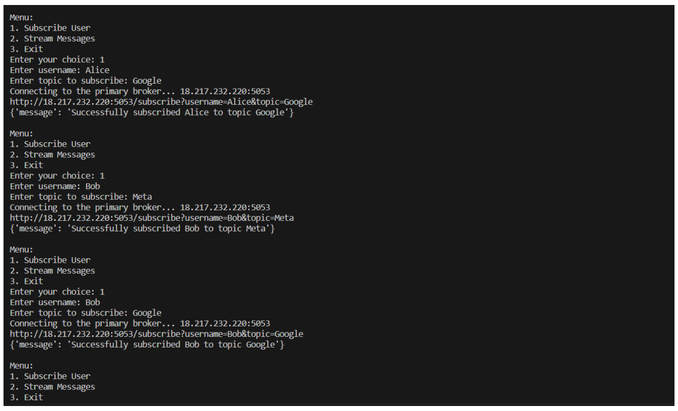
Snapshot:

Publisher:



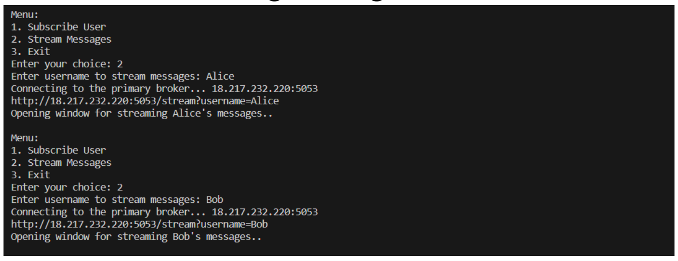
```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Google
Enter data: Hello Google
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic Google data Hello Google'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Meta
Enter data: Hiring Meta
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic Meta data Hiring Meta'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Meta
Enter data: Social event
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic Meta data Social event'}
```

Subscriber:



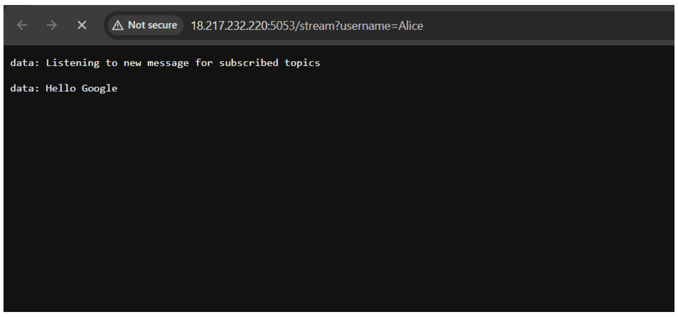
```
Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Alice
Enter topic to subscribe: Google
Connecting to the primary broker... 18.217.232.220:5053
http://18.217.232.220:5053/subscribe?username=Alice&topic=Google
{'message': 'Successfully subscribed Alice to topic Google'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Bob
Enter topic to subscribe: Meta
Connecting to the primary broker... 18.217.232.220:5053
http://18.217.232.220:5053/subscribe?username=Bob&topic=Meta
{'message': 'Successfully subscribed Bob to topic Meta'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Bob
Enter topic to subscribe: Google
Connecting to the primary broker... 18.217.232.220:5053
http://18.217.232.220:5053/subscribe?username=Bob&topic=Google
{'message': 'Successfully subscribed Bob to topic Google'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
```

Subscribers streaming messages:

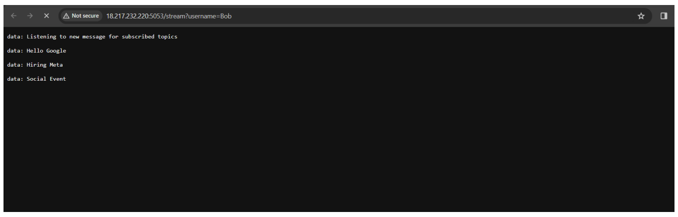


```
Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 2
Enter username to stream messages: Alice
Connecting to the primary broker... 18.217.232.220:5053
http://18.217.232.220:5053/stream?username=Alice
Opening window for streaming Alice's messages..

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 2
Enter username to stream messages: Bob
Connecting to the primary broker... 18.217.232.220:5053
http://18.217.232.220:5053/stream?username=Bob
Opening window for streaming Bob's messages..
```

Streaming window for Alice and Bob:



```
data: Listening to new message for subscribed topics

data: Hello Google
```


18.217.232.220:5053/stream?username=Bob

```
data: Listening to new message for subscribed topics

data: Hello Google

data: Hiring Meta

data: Social Event
```

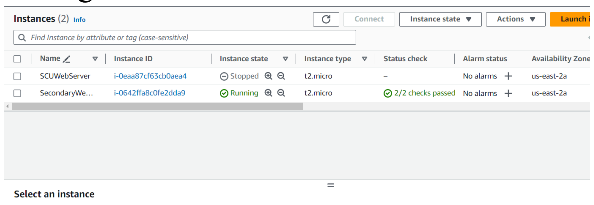
System Scenario II(Failure of Primary):

- Continuation of scenario I where Primary goes down.
- Publisher:
  - Google
    - Family Event
    - Employee
  - Tiktok
    - Hi Tiktok
- Subscriber:
  - Carlos
    - Topic : Tiktok
  - Bob
    - Topic : Google
  - Alice
    - Topic : Google

Results:

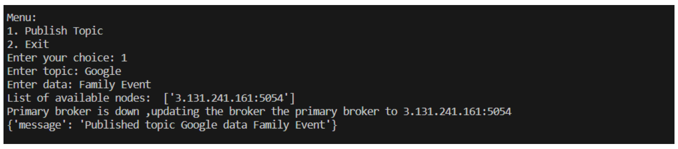
All the subscribers are seamlessly able to consume their messages via the secondary message broker which is now acting as a primary broker.

Switching off the ec2 node to make primary message broker failure

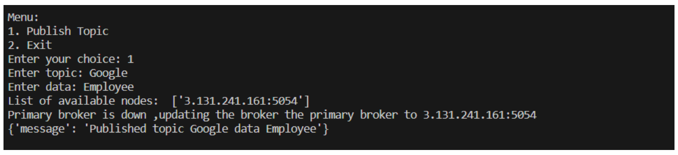

Instances (2) Info

	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
☐	SCUWebServer	i-0eaa87cf63cb0aea4	Stopped	t2.micro	–	No alarms	us-east-2a
☐	SecondaryWe...	i-0642ffa8c0fe2dda9	Running	t2.micro	2/2 checks passed	No alarms	us-east-2a

- Alice listening messages(GET stream): Family Event, Employee
  (Explanation:) Subscriber Alice redirected to Secondary broker with ip 3.131.241.161*(as it becomes new primary if actual primary is down)*
  Publisher:

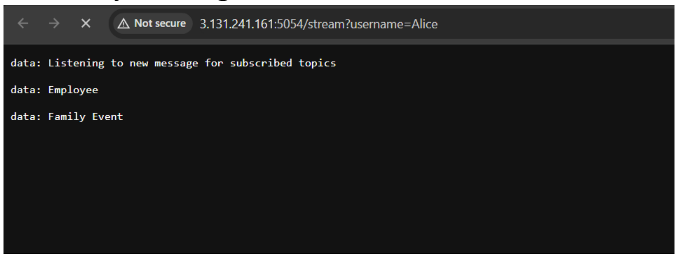


```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Google
Enter data: Family Event
List of available nodes:  ['3.131.241.161:5054']
Primary broker is down ,updating the broker the primary broker to 3.131.241.161:5054
{'message': 'Published topic Google data Family Event'}
```



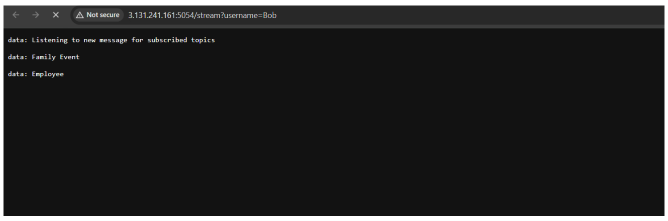
```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Google
Enter data: Employee
List of available nodes:  ['3.131.241.161:5054']
Primary broker is down ,updating the broker the primary broker to 3.131.241.161:5054
{'message': 'Published topic Google data Employee'}
```

Alice able to stream her messages via the secondary message broker:


3.131.241.161:5054/stream?username=Alice

```
data: Listening to new message for subscribed topics

data: Employee

data: Family Event
```

- Bob listening messages (GET stream): Family Event, Employee
  (Explanation:) Subscriber Bob redirected to Secondary broker(as it becomes new primary if actual primary is down)

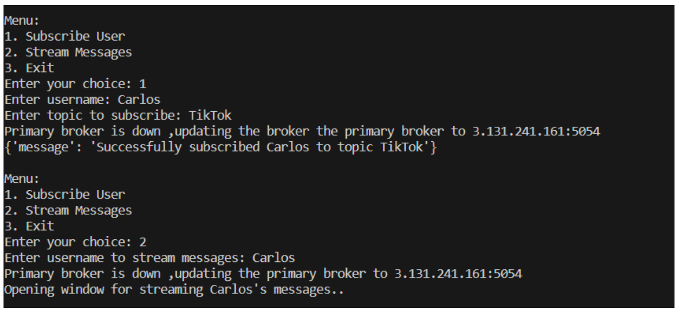

3.131.241.161:5054/stream?username=Bob

```
data: Listening to new message for subscribed topics

data: Family Event

data: Employee
```

- New subscriber Carlos listening messages(GET stream): Hi Tiktok

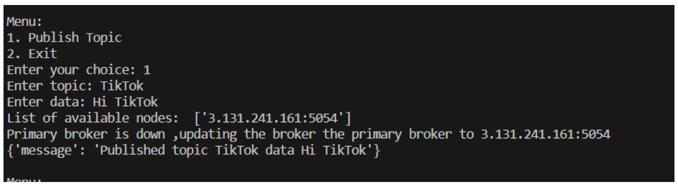
Snapshot:

Subscriber:



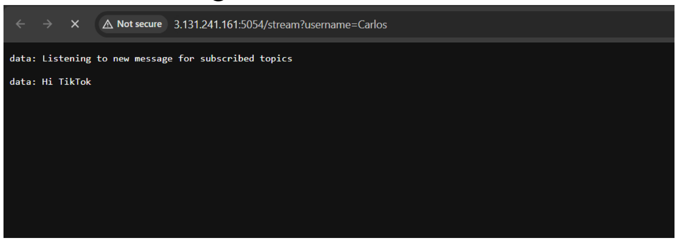
```
Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Carlos
Enter topic to subscribe: TikTok
Primary broker is down ,updating the broker the primary broker to 3.131.241.161:5054
{'message': 'Successfully subscribed Carlos to topic TikTok'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 2
Enter username to stream messages: Carlos
Primary broker is down ,updating the primary broker to 3.131.241.161:5054
Opening window for streaming Carlos's messages..
```

Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: TikTok
Enter data: Hi TikTok
List of available nodes:  ['3.131.241.161:5054']
Primary broker is down ,updating the broker the primary broker to 3.131.241.161:5054
{'message': 'Published topic TikTok data Hi TikTok'}
```

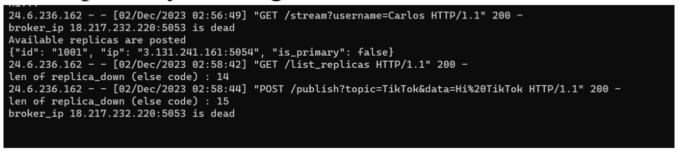
Streaming window:



```
3.131.241.161:5054/stream?username=Carlos

data: Listening to new message for subscribed topics

data: Hi TikTok
```

Heartbeat exchange:

Secondary message broker detecting that the primary message broker is down and updating itself as the primary message broker.



```
24.6.236.162 - - [02/Dec/2023 02:56:49] "GET /stream?username=Carlos HTTP/1.1" 200 -
broker_ip 18.217.232.220:5053 is dead
Available replicas are posted
{"id": "1001", "ip": "3.131.241.161:5054", "is_primary": false}
24.6.236.162 - - [02/Dec/2023 02:58:42] "GET /list_replicas HTTP/1.1" 200 -
len of replica_down (else code) : 14
24.6.236.162 - - [02/Dec/2023 02:58:44] "POST /publish?topic=TikTok&data=Hi%20TikTok HTTP/1.1" 200 -
len of replica_down (else code) : 15
broker_ip 18.217.232.220:5053 is dead
```

System Scenario III (Primary is started again, i.e. Primary + Secondary both available):

- Continuation of scenario II where Primary restores its state after coming up.
- After the first heartbeat of primary towards secondary, secondary restores primary state to consistent to secondary itself(using replication protocol) and rest operations are handled at primary.
- Snapshot(Restoring Primary state with subscribed user addition and other state change when it was down):

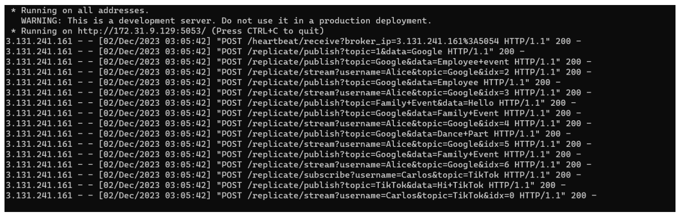
- Publisher:
    - Tiktok
        - Hi Online

Results:

- Carlos listening messages(GET stream): Hi Dance

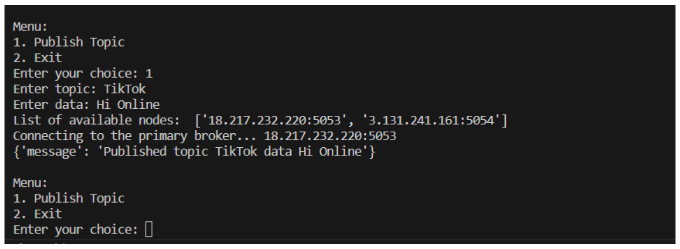
Snapshot:

Primary message broker restoring its state



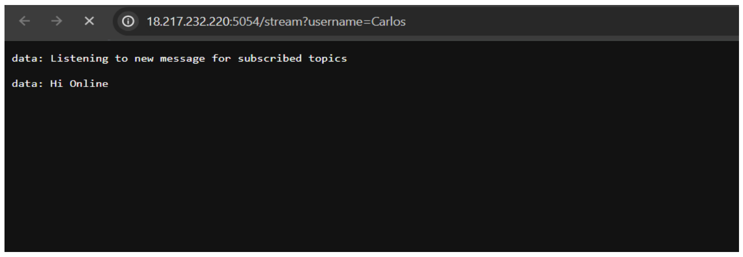
```
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.31.9.129:5053/ (Press CTRL+C to quit)
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=1&data=Google HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Employee+event HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=2 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Employee HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=3 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Family+Event&data=Hello HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Family+Event HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=4 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Dance+Part HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=5 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Family+Event HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=6 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/subscribe?username=Carlos&topic=TikTok HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=TikTok&data=Hi+TikTok HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Carlos&topic=TikTok&idx=0 HTTP/1.1" 200 -
```

Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: TikTok
Enter data: Hi Online
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic TikTok data Hi Online'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 
```

Successfully streaming messages by restoring the previous states:



```
18.217.232.220:5054/stream?username=Carlos

data: Listening to new message for subscribed topics

data: Hi Online
```

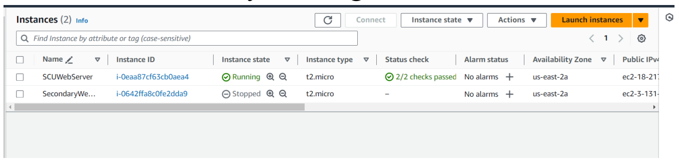
System Scenario IV (Failure of Secondary):

- Continuation of scenario III where Secondary fails.
- Publisher:
    - Tiktok
        - Hi Event

Results:

Snapshot:

Failure of secondary message broker:

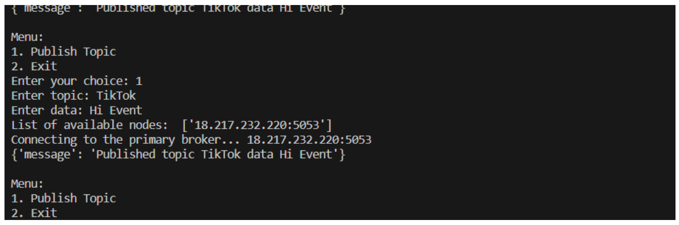


Instances (2) Info

Find Instance by attribute or tag (case-sensitive)

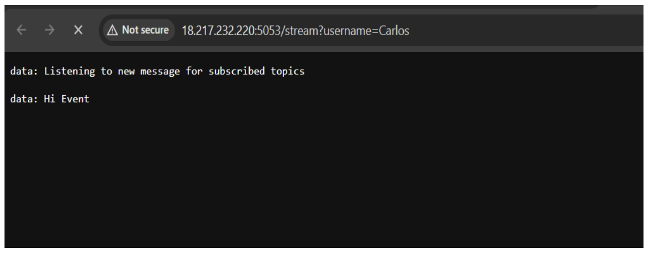
	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv
☐	SCUWebServer	i-0eaa87cf63cb0aea4	Running	t2.micro	2/2 checks passed	No alarms	us-east-2a	ec2-18-21
☐	SecondaryWe...	i-0642ffa8c0fe2dda9	Stopped	t2.micro	–	No alarms	us-east-2a	ec2-3-131-

Publisher:



```
{'message': 'Published topic TikTok data Hi Event'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: TikTok
Enter data: Hi Event
List of available nodes:  ['18.217.232.220:5053']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic TikTok data Hi Event'}

Menu:
1. Publish Topic
2. Exit
```

- Carlos listening messages(GET stream): Hi Event



```
data: Listening to new message for subscribed topics

data: Hi Event
```

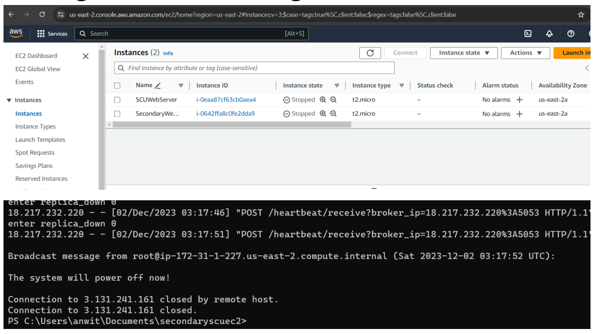
System Scenario V (Primary and Secondary both fails)

- Fail both the message brokers..
- Restart both message brokers
- Publisher:
  - Tiktok
    - Hi You!

Results:

- Carol listening messages(GET stream): Hi You!
  (Explanation:) When both goes down system itself restores its state and already knows that Carol is subscribed to topic tiktok so once connection for listening is established via GET stream publishers and subscribers do not need to rework

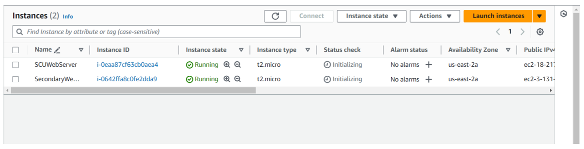
  Failing both the message brokers:



Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
SCUWebServer	i-0eaa87cf63cb0aea4	Stopped	t2.micro	–	No alarms	us-east-2a
SecondaryWe...	i-0642ffa8c0fe2dda9	Stopped	t2.micro	–	No alarms	us-east-2a

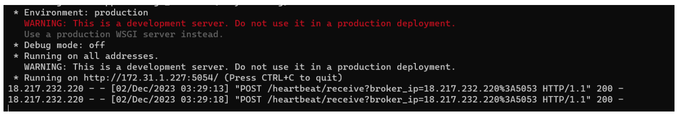
```
enter replica_down 0
18.217.232.220 - - [02/Dec/2023 03:17:46] "POST /heartbeat/receive?broker_ip=18.217.232.220%3A5053 HTTP/1.1
enter replica_down 0
18.217.232.220 - - [02/Dec/2023 03:17:51] "POST /heartbeat/receive?broker_ip=18.217.232.220%3A5053 HTTP/1.1

Broadcast message from root@ip-172-31-1-227.us-east-2.compute.internal (Sat 2023-12-02 03:17:52 UTC):

The system will power off now!

Connection to 3.131.241.161 closed by remote host.
Connection to 3.131.241.161 closed.
PS C:\Users\anwit\Documents\secondaryscuec2>
```

  Restart both the message brokers:

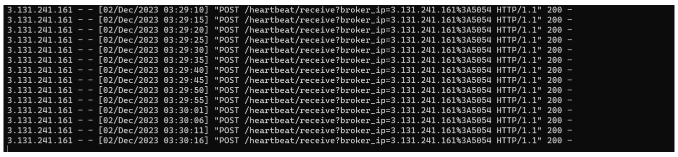


Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv
SCUWebServer	i-0eaa87cf63cb0aea4	Running	t2.micro	Initializing	No alarms	us-east-2a	ec2-18-21
SecondaryWe...	i-0642ffa8c0fe2dda9	Running	t2.micro	Initializing	No alarms	us-east-2a	ec2-3-131-

Both the message brokers are able to restore their states:

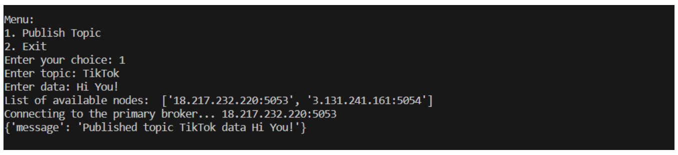


```
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.31.1.227:5054/ (Press CTRL+C to quit)
18.217.232.220 - - [02/Dec/2023 03:29:13] "POST /heartbeat/receive?broker_ip=18.217.232.220%3A5053 HTTP/1.1" 200 -
18.217.232.220 - - [02/Dec/2023 03:29:18] "POST /heartbeat/receive?broker_ip=18.217.232.220%3A5053 HTTP/1.1" 200 -
```



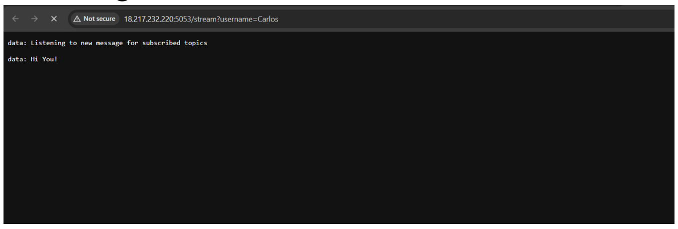
```
3.131.241.161 - - [02/Dec/2023 03:29:10] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:15] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:20] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:25] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:30] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:35] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:40] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:45] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:50] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:55] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:01] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:06] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:11] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:16] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
```

Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: TikTok
Enter data: Hi You!
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic TikTok data Hi You!'}
```

Streaming window:

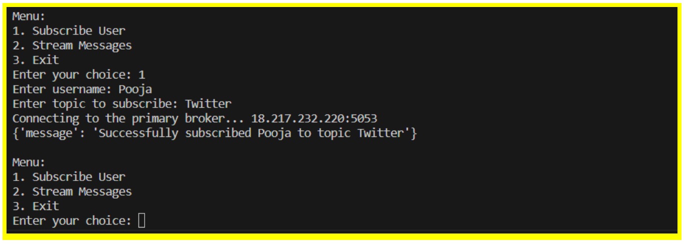


```
data: Listening to new message for subscribed topics

data: Hi You!
```

System Scenario VI(Subscriber offline/Client Failure):

- Subscriber:
  - Username: Pooja
  - Topic : Twitter
- Publisher:
  - Topic : Twitter
    - Hello tweets

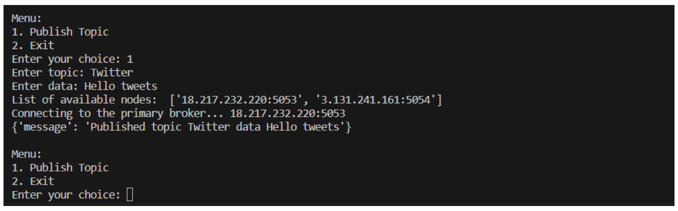
Result: Subscriber is offline while the publisher has published the data and able to stream the left off messages after she is back online .
Snapshot:
Subscriber subscribed to the topic but has not streamed her messages:



```
Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Pooja
Enter topic to subscribe: Twitter
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Successfully subscribed Pooja to topic Twitter'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 
```

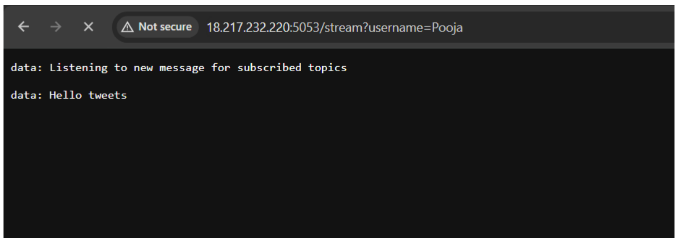
Here, Pooja has just subscribed to the topic but not yet streamed her messages which implies that she is offline.

Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Twitter
Enter data: Hello tweets
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic Twitter data Hello tweets'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 
```

Subscriber able to stream her messages when she is back online:



```
Not secure  18.217.232.220:5053/stream?username=Pooja

data: Listening to new message for subscribed topics

data: Hello tweets
```

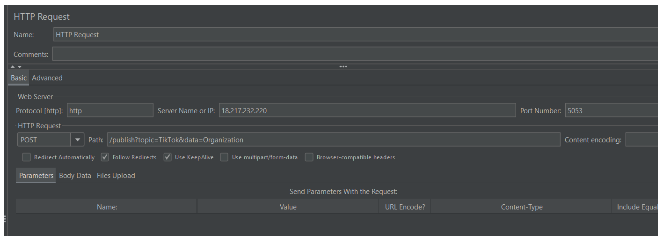
IX. Performance Analysis

The performance of the system is assessed via the jmeter tool.We were able to achieve the optimal results in the performance testing:

Publisher module:

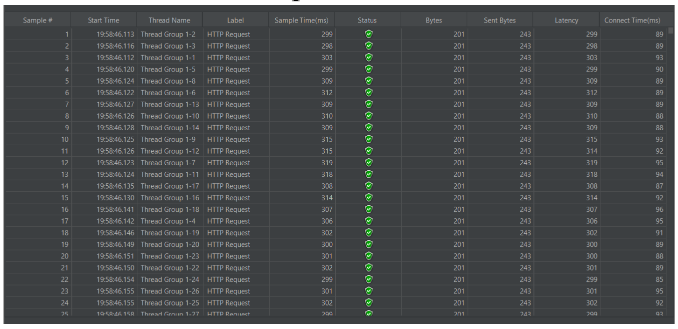
Expected Performance:
Avg Latency Expected: 200-250 ms
Avg Throughout Expected: 80-90ms

Actual Performance:



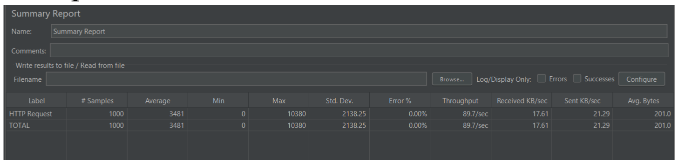
```
HTTP Request
Name: HTTP Request
Comments:
Basic  Advanced
Web Server
Protocol [http]: http    Server Name or IP: 18.217.232.220    Port Number: 5053
HTTP Request
POST    Path: /publish?topic=TikTok&data=Organization    Content encoding:
[ ] Redirect Automatically  [✓] Follow Redirects  [✓] Use KeepAlive  [ ] Use multipart/form-data  [ ] Browser-compatible headers
Parameters  Body Data  Files Upload
Send Parameters With the Request:
Name:    Value    URL Encode?    Content-Type    Include Equal
```

Number of concurrent publishers:1000



Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	19:58:46.113	Thread Group 1-2	HTTP Request	299	✓	201	243	299	89
2	19:58:46.116	Thread Group 1-3	HTTP Request	298	✓	201	243	298	89
3	19:58:46.112	Thread Group 1-1	HTTP Request	303	✓	201	243	303	93
4	19:58:46.120	Thread Group 1-5	HTTP Request	299	✓	201	243	299	90
5	19:58:46.124	Thread Group 1-8	HTTP Request	309	✓	201	243	309	89
6	19:58:46.122	Thread Group 1-6	HTTP Request	312	✓	201	243	312	89
7	19:58:46.127	Thread Group 1-13	HTTP Request	309	✓	201	243	309	89
8	19:58:46.126	Thread Group 1-10	HTTP Request	310	✓	201	243	310	88
9	19:58:46.128	Thread Group 1-14	HTTP Request	309	✓	201	243	309	88
10	19:58:46.125	Thread Group 1-9	HTTP Request	315	✓	201	243	315	93
11	19:58:46.126	Thread Group 1-12	HTTP Request	315	✓	201	243	314	92
12	19:58:46.123	Thread Group 1-7	HTTP Request	319	✓	201	243	319	95
13	19:58:46.124	Thread Group 1-11	HTTP Request	318	✓	201	243	318	94
14	19:58:46.135	Thread Group 1-17	HTTP Request	308	✓	201	243	308	87
15	19:58:46.130	Thread Group 1-16	HTTP Request	314	✓	201	243	314	92
16	19:58:46.141	Thread Group 1-18	HTTP Request	307	✓	201	243	307	96
17	19:58:46.142	Thread Group 1-4	HTTP Request	306	✓	201	243	306	95
18	19:58:46.146	Thread Group 1-19	HTTP Request	302	✓	201	243	302	91
19	19:58:46.149	Thread Group 1-20	HTTP Request	300	✓	201	243	300	89
20	19:58:46.151	Thread Group 1-23	HTTP Request	301	✓	201	243	300	88
21	19:58:46.150	Thread Group 1-22	HTTP Request	302	✓	201	243	301	89
22	19:58:46.154	Thread Group 1-24	HTTP Request	299	✓	201	243	299	85
23	19:58:46.155	Thread Group 1-26	HTTP Request	301	✓	201	243	301	95
24	19:58:46.155	Thread Group 1-25	HTTP Request	302	✓	201	243	302	92
25	19:58:46.158	Thread Group 1-27	HTTP Request	299	✓	201	243	299	93

Summary report of the performance metrics for publisher module:
1000 publisher:



Summary Report
Name: Summary Report
Comments:
Write results to file / Read from file
Filename   Log/Display Only: [ ] Errors [ ] Successes

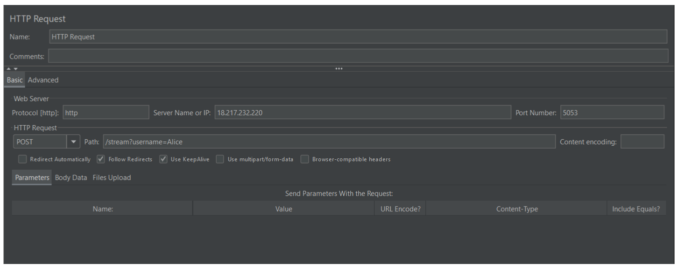
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	3481	0	10380	2138.25	0.00%	89.7/sec	17.61	21.29	201.0
TOTAL	1000	3481	0	10380	2138.25	0.00%	89.7/sec	17.61	21.29	201.0

The latency and throughput achieved for the system were optimal for the system with 0% error rate.

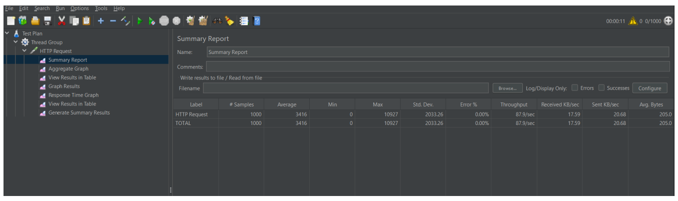
Subscriber module:
Number of concurrent users: 1000

Expected Performance:
Avg Latency Expected: 200-250 ms
Avg Throughout Expected: 85-90ms

Actual Performance:

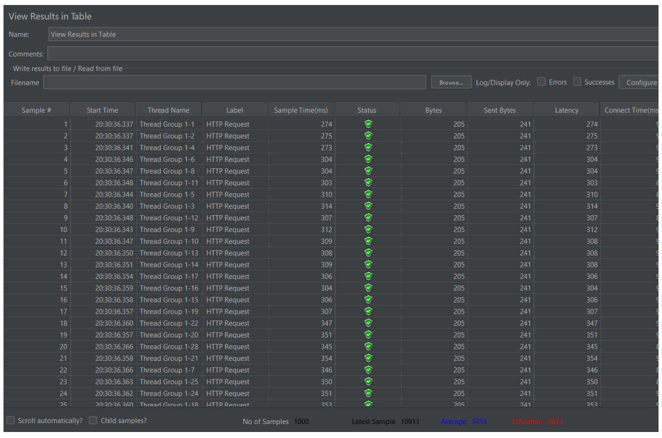


```
HTTP Request
Name: HTTP Request
Comments:
Basic  Advanced
Web Server
Protocol [http]: http    Server Name or IP: 18.217.232.220    Port Number: 5053
HTTP Request
POST    Path: /stream?username=Alice    Content encoding:
[ ] Redirect Automatically  [✓] Follow Redirects  [✓] Use KeepAlive  [ ] Use multipart/form-data  [ ] Browser-compatible headers
Parameters  Body Data  Files Upload
Send Parameters With the Request:
Name:    Value    URL Encode?    Content-Type    Include Equals?
```



Summary Report
Name: Summary Report
Comments:
Write results to file / Read from file
Filename   Log/Display Only: [ ] Errors [ ] Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	3416	0	10927	2033.26	0.00%	87.9/sec	17.59	20.68	205.0
TOTAL	1000	3416	0	10927	2033.26	0.00%	87.9/sec	17.59	20.68	205.0



View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file

Filename

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms
1	20:30:36.337	Thread Group 1-1	HTTP Request	274	✓	205	241	274	9
2	20:30:36.337	Thread Group 1-2	HTTP Request	275	✓	205	241	275	9
3	20:30:36.341	Thread Group 1-4	HTTP Request	273	✓	205	241	273	9
4	20:30:36.346	Thread Group 1-6	HTTP Request	304	✓	205	241	304	9
5	20:30:36.347	Thread Group 1-8	HTTP Request	304	✓	205	241	304	9
6	20:30:36.348	Thread Group 1-11	HTTP Request	303	✓	205	241	303	8
7	20:30:36.344	Thread Group 1-5	HTTP Request	310	✓	205	241	310	8
8	20:30:36.340	Thread Group 1-3	HTTP Request	314	✓	205	241	314	9
9	20:30:36.348	Thread Group 1-12	HTTP Request	307	✓	205	241	307	8
10	20:30:36.343	Thread Group 1-9	HTTP Request	312	✓	205	241	312	9
11	20:30:36.347	Thread Group 1-10	HTTP Request	309	✓	205	241	308	9
12	20:30:36.350	Thread Group 1-13	HTTP Request	308	✓	205	241	308	9
13	20:30:36.351	Thread Group 1-14	HTTP Request	309	✓	205	241	308	9
14	20:30:36.354	Thread Group 1-17	HTTP Request	306	✓	205	241	306	9
15	20:30:36.359	Thread Group 1-16	HTTP Request	304	✓	205	241	304	9
16	20:30:36.358	Thread Group 1-15	HTTP Request	306	✓	205	241	306	9
17	20:30:36.357	Thread Group 1-19	HTTP Request	307	✓	205	241	307	9
18	20:30:36.360	Thread Group 1-22	HTTP Request	347	✓	205	241	347	9
19	20:30:36.357	Thread Group 1-20	HTTP Request	351	✓	205	241	351	9
20	20:30:36.366	Thread Group 1-28	HTTP Request	345	✓	205	241	345	8
21	20:30:36.358	Thread Group 1-21	HTTP Request	354	✓	205	241	354	9
22	20:30:36.366	Thread Group 1-7	HTTP Request	346	✓	205	241	346	8
23	20:30:36.363	Thread Group 1-25	HTTP Request	350	✓	205	241	350	8
24	20:30:36.362	Thread Group 1-24	HTTP Request	351	✓	205	241	351	9
25	20:30:36.360	Thread Group 1-18	HTTP Request	353	✓	205	241	353	9

Scroll automatically? Child samples? No of Samples 1000 Latest Sample 10913 Average 3416 Deviation 2033

The subscriber module is providing the optimal performance and is capable to handling multiple concurrent subscription requests.

X. Results

Test Cases:

Test Case I:

- Both primary and Secondary broker are alive
- Subscribers:
  - Username: Alice
    - topics: Google
  - Username: Bob
    - topics: Google, Meta

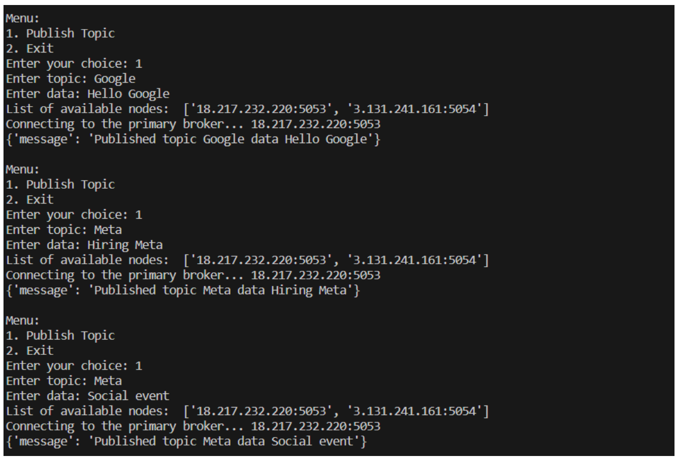
- Publishers:
  - Topic: Google
    - Hiring Google
  - Topic: Meta
    - Hiring Meta
    - Social Event

Results:

- Alice Listening messages(GET stream): Hiring Google
- Bob Listening messages(GET stream): Hiring Google, Hiring Meta, Social Event

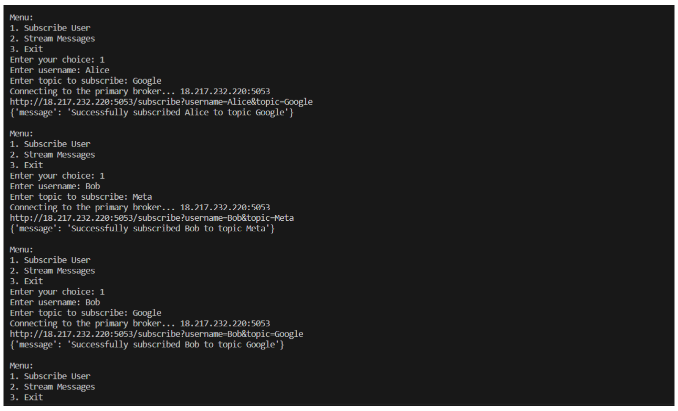
Snapshot:

Publisher:



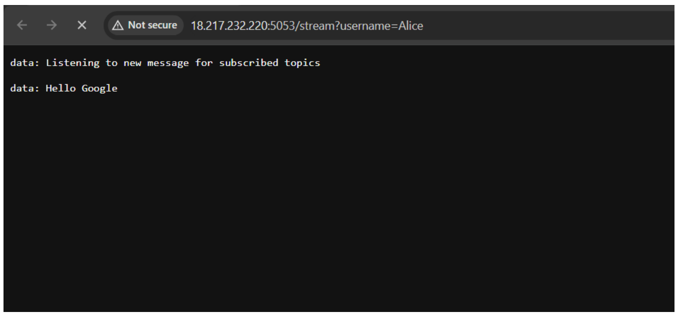
```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Google
Enter data: Hello Google
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic Google data Hello Google'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Meta
Enter data: Hiring Meta
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic Meta data Hiring Meta'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Meta
Enter data: Social event
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic Meta data Social event'}
```

Subscriber:



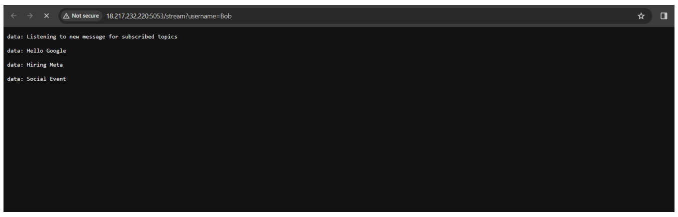
```
Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Alice
Enter topic to subscribe: Google
Connecting to the primary broker... 18.217.232.220:5053
http://18.217.232.220:5053/subscribe?username=Alice&topic=Google
{'message': 'Successfully subscribed Alice to topic Google'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Bob
Enter topic to subscribe: Meta
Connecting to the primary broker... 18.217.232.220:5053
http://18.217.232.220:5053/subscribe?username=Bob&topic=Meta
{'message': 'Successfully subscribed Bob to topic Meta'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Bob
Enter topic to subscribe: Google
Connecting to the primary broker... 18.217.232.220:5053
http://18.217.232.220:5053/subscribe?username=Bob&topic=Google
{'message': 'Successfully subscribed Bob to topic Google'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
```

Streaming window for Alice and Bob:



Not secure 18.217.232.220:5053/stream?username=Alice

```
data: Listening to new message for subscribed topics

data: Hello Google
```



Not secure 18.217.232.220:5053/stream?username=Bob

```
data: Listening to new message for subscribed topics

data: Hello Google

data: Hiring Meta

data: Social Event
```

Test Case II(Failure of Primary):

- Continuation of scenario I where Primary goes down.
- Publisher:

    - Google
        - Dance party
        - Family day
    - Tiktok
        - Hi Tiktok
- Subscriber:
    - Carol
        - Topic : Tiktok

Results:

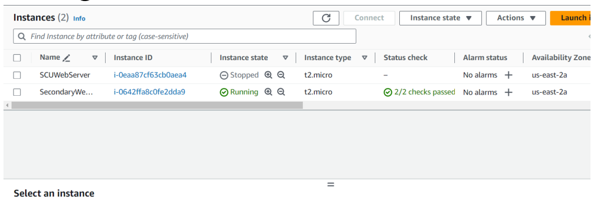
- Alice listening messages(GET stream): Dance Part, Family Event
  (Explanation:) Subscriber Alice redirected to Secondary broker(as it becomes new primary if actual primary is down)

- Bob listening messages (GET stream):Employee, Family Event
  (Explanation:) Subscriber Bob redirected to Secondary broker(as it becomes new primary if actual primary is down)

- Carol listening messages(GET stream): Hi Tiktok

Snapshot:

Switching off the ec2 node to make primary message broker failure



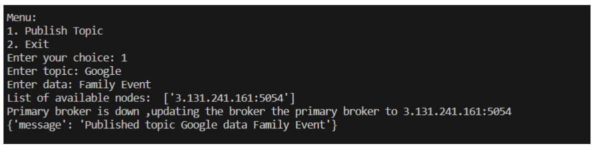
Instances (2) Info

Find Instance by attribute or tag (case-sensitive)

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
SCUWebServer	i-0eaa87cf63cb0aea4	Stopped	t2.micro	–	No alarms	us-east-2a
SecondaryWe...	i-0642ffa8c0fe2dda9	Running	t2.micro	2/2 checks passed	No alarms	us-east-2a

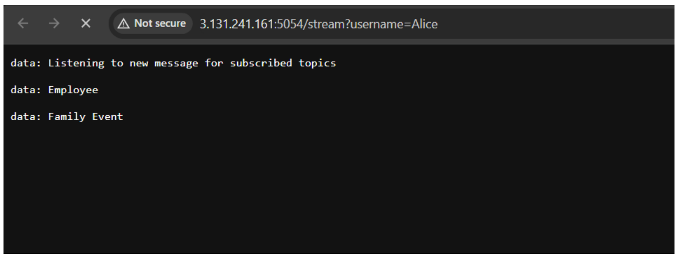
Select an instance

- Alice listening messages(GET stream): Family Event, Employee
  (Explanation:) Subscriber Alice redirected to Secondary broker(as it becomes new primary if actual primary is down)
  Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Google
Enter data: Family Event
List of available nodes:  ['3.131.241.161:5054']
Primary broker is down ,updating the broker the primary broker to 3.131.241.161:5054
{'message': 'Published topic Google data Family Event'}
```

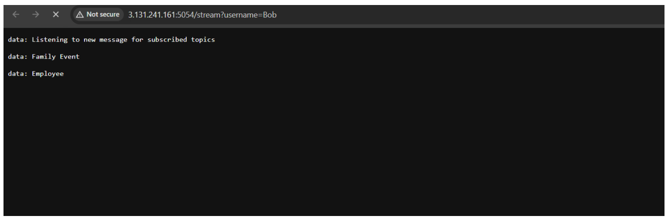
Alice able to stream her messages via the secondary message broker:



3.131.241.161:5054/stream?username=Alice

```
data: Listening to new message for subscribed topics

data: Employee

data: Family Event
```

- Bob listening messages (GET stream): Family Event, Employee
  (Explanation:) Subscriber Bob redirected to Secondary broker(as it becomes new primary if actual primary is down)



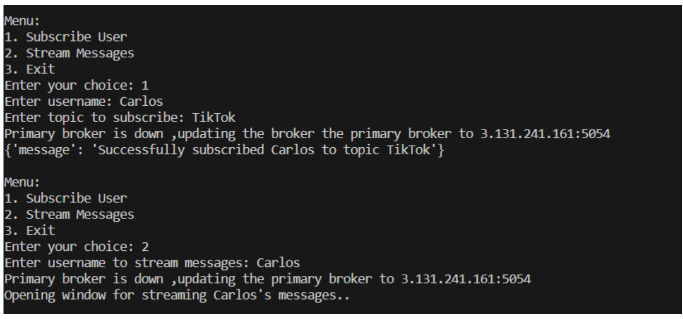
3.131.241.161:5054/stream?username=Bob

```
data: Listening to new message for subscribed topics

data: Family Event

data: Employee
```

- New subscriber Carlos listening messages(GET stream): Hi Tiktok

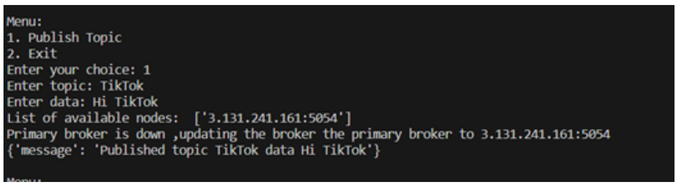
Snapshot:

Subscriber:



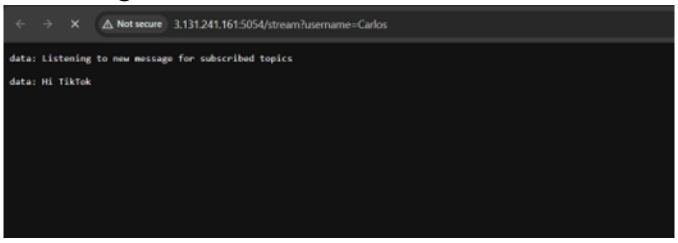
```
Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Carlos
Enter topic to subscribe: TikTok
Primary broker is down ,updating the broker the primary broker to 3.131.241.161:5054
{'message': 'Successfully subscribed Carlos to topic TikTok'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 2
Enter username to stream messages: Carlos
Primary broker is down ,updating the primary broker to 3.131.241.161:5054
Opening window for streaming Carlos's messages..
```

Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: TikTok
Enter data: Hi TikTok
List of available nodes:  ['3.131.241.161:5054']
Primary broker is down ,updating the broker the primary broker to 3.131.241.161:5054
{'message': 'Published topic TikTok data Hi TikTok'}
```

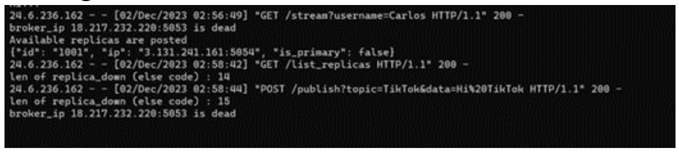
Streaming window:



3.131.241.161:5054/stream?username=Carlos

```
data: Listening to new message for subscribed topics

data: Hi TikTok
```

Heartbeat exchange:
Secondary message broker detecting that the primary message
broker is down and updating itself as the primary message broker.



```
24.6.236.162 - - [02/Dec/2023 02:56:49] "GET /stream?username=Carlos HTTP/1.1" 200 -
broker_ip 18.217.232.220:5053 is dead
Available replicas are posted
{"id": "1001", "ip": "3.131.241.161:5054", "is_primary": false}
24.6.236.162 - - [02/Dec/2023 02:58:42] "GET /list_replicas HTTP/1.1" 200 -
len of replica_down (else code) : 14
24.6.236.162 - - [02/Dec/2023 02:58:44] "POST /publish?topic=TikTok&data=Hi%20TikTok HTTP/1.1" 200 -
len of replica_down (else code) : 15
broker_ip 18.217.232.220:5053 is dead
```

Test Case III (Primary is started again, i.e. Primary + Secondary both available):

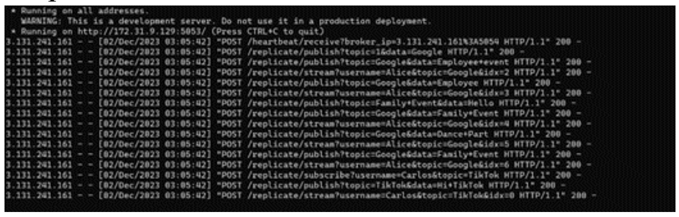
- Continuation of scenario II where Primary restores its state after coming up.
- After the first heartbeat of primary towards secondary, secondary restores primary state to consistent to secondary itself(using replication protocol) and rest operations are handled at primary.
- Snapshot(Restoring Primary state with subscribed user addition and other state change when it was down):

- Publisher:
    - Tiktok
        - Hi Online

Results:

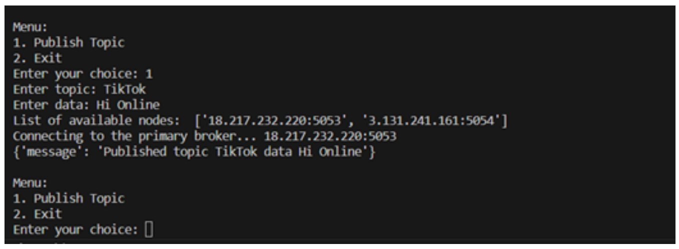
- Carlos listening messages(GET stream): Hi Dance

Snapshot:



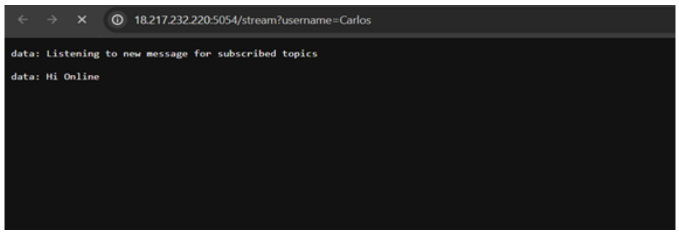
```
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.31.9.129:5053/ (Press CTRL+C to quit)
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=1&data=Google HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Employee+event HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=2 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Employee HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=3 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Family+Event&data=Hello HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Family+Event HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=4 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Dance+Part HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=5 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=Google&data=Family+Event HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Alice&topic=Google&idx=6 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/subscribe?username=Carlos&topic=TikTok HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/publish?topic=TikTok&data=Hi+TikTok HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:05:42] "POST /replicate/stream?username=Carlos&topic=TikTok&idx=0 HTTP/1.1" 200 -
```

Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: TikTok
Enter data: Hi Online
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic TikTok data Hi Online'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: []
```

Successfully streaming messages by restoring the previous states:



18.217.232.220:5054/stream?username=Carlos

```
data: Listening to new message for subscribed topics

data: Hi Online
```

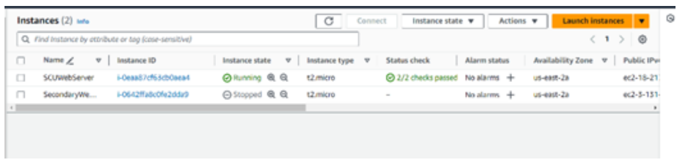
Test Case IV (Failure of Secondary):

- Continuation of scenario III where Secondary fails.
- Publisher:
    - Tiktok
        - Hi Event

Results:

- Carol listening messages(GET stream): Hi Online

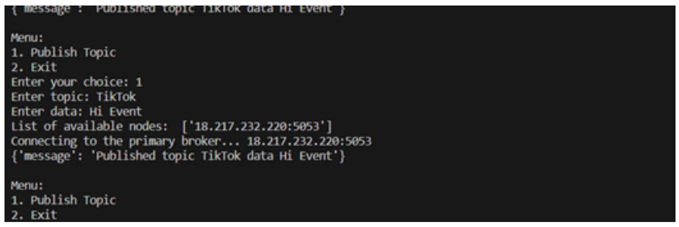
Snapshot:



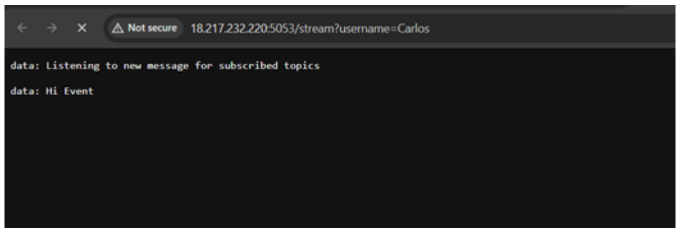
Instances (2) Info

	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv
	SCUWebServer	i-0eaa87cf63cb0aea4	Running	t2.micro	2/2 checks passed	No alarms	us-east-2a	ec2-18-21
	SecondaryWe...	i-0642ffa8c0fe2dda9	Stopped	t2.micro	-	No alarms	us-east-2a	ec2-3-131-

Publisher:



```
{'message': 'Published topic TikTok data Hi Event'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: TikTok
Enter data: Hi Event
List of available nodes:  ['18.217.232.220:5053']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic TikTok data Hi Event'}

Menu:
1. Publish Topic
2. Exit
```



Not secure 18.217.232.220:5053/stream?username=Carlos

```
data: Listening to new message for subscribed topics

data: Hi Event
```

Test Case V (Primary and Secondary both fails)

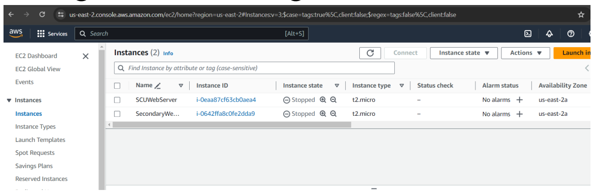
- Continuation of scenario IV where Primary also fails.
- We restart our brokers/broker
- Publisher:
    - Tiktok
        - Hi You!

Results:

- Carol listening messages(GET stream): Hi You!
  (Explanation:) When both goes down system itself restores its state and already knows that Carol is subscribed to topic tiktok so once connection for listening is established via GET stream publishers and subscribers do not need to rework

Snapshots:

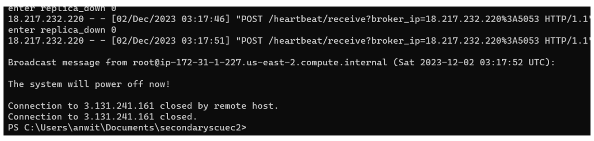
Failing both the message brokers:



us-east-2.console.aws.amazon.com/ec2/home?region=us-east-2#Instances:v=3;$case=tags:true%5C,client:false;$regex=tags:false%5C,client:false

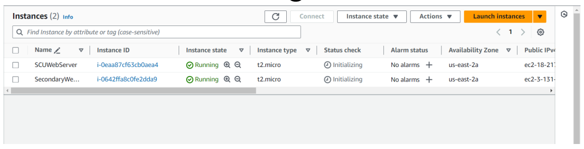
Instances (2)

	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
☐	SCUWebServer	i-0eaa87cf63cb0aea4	Stopped	t2.micro	–	No alarms	us-east-2a
☐	SecondaryWe...	i-0642ffa8c0fe2dda9	Stopped	t2.micro	–	No alarms	us-east-2a



```
enter replica_down 0
18.217.232.220 - - [02/Dec/2023 03:17:46] "POST /heartbeat/receive?broker_ip=18.217.232.220%3A5053 HTTP/1.1"
enter replica_down 0
18.217.232.220 - - [02/Dec/2023 03:17:51] "POST /heartbeat/receive?broker_ip=18.217.232.220%3A5053 HTTP/1.1"

Broadcast message from root@ip-172-31-1-227.us-east-2.compute.internal (Sat 2023-12-02 03:17:52 UTC):

The system will power off now!

Connection to 3.131.241.161 closed by remote host.
Connection to 3.131.241.161 closed.
PS C:\Users\anwit\Documents\secondaryscuec2>
```

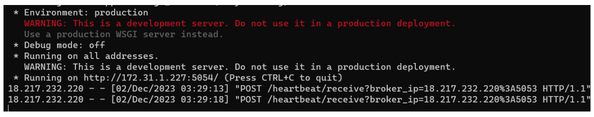
Restart both the message brokers:



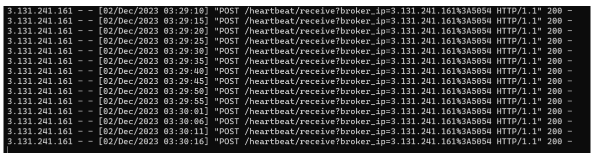
Instances (2)

	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv
☐	SCUWebServer	i-0eaa87cf63cb0aea4	Running	t2.micro	Initializing	No alarms	us-east-2a	ec2-18-21
☐	SecondaryWe...	i-0642ffa8c0fe2dda9	Running	t2.micro	Initializing	No alarms	us-east-2a	ec2-3-131-

Both the message brokers are able to restore their states:

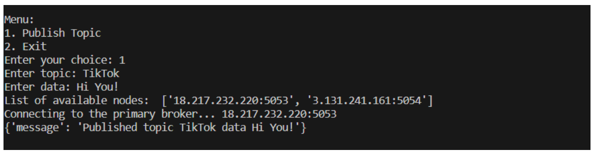


```
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.31.1.227:5054/ (Press CTRL+C to quit)
18.217.232.220 - - [02/Dec/2023 03:29:13] "POST /heartbeat/receive?broker_ip=18.217.232.220%3A5053 HTTP/1.1"
18.217.232.220 - - [02/Dec/2023 03:29:18] "POST /heartbeat/receive?broker_ip=18.217.232.220%3A5053 HTTP/1.1"
```



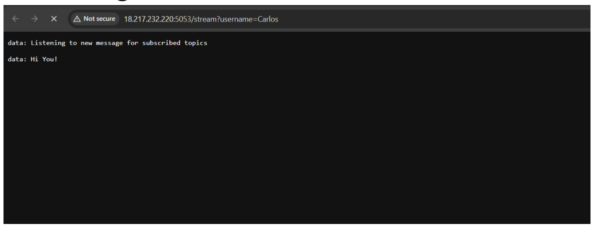
```
3.131.241.161 - - [02/Dec/2023 03:29:10] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:15] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:20] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:25] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:30] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:35] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:40] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:45] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:50] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:55] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:01] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:06] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:11] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:16] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
```

Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: TikTok
Enter data: Hi You!
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic TikTok data Hi You!'}
```

Streaming window:



Not secure 18.217.232.220:5053/stream?username=Carlos

```
data: Listening to new message for subscribed topics

data: Hi You!
```

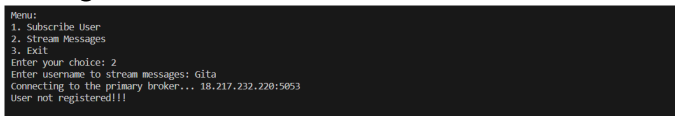
Test Case VI (Security : User not Registered):

- Subscriber: Gita (Not subscribed to any topic)
  - Attempts to listen:

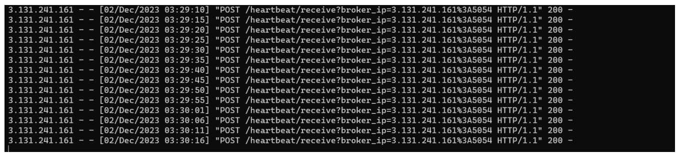
Results:

Results:

- Ensuring security of the system by not allowing unregistered users to stream the messages.



```
Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 2
Enter username to stream messages: Gita
Connecting to the primary broker... 18.217.232.220:5053
User not registered!!!
```



```
3.131.241.161 - - [02/Dec/2023 03:29:10] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:15] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:20] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:25] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:30] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:35] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:40] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:45] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:50] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:29:55] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:01] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:06] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:11] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
3.131.241.161 - - [02/Dec/2023 03:30:16] "POST /heartbeat/receive?broker_ip=3.131.241.161%3A5054 HTTP/1.1" 200 -
```

Snapshots:

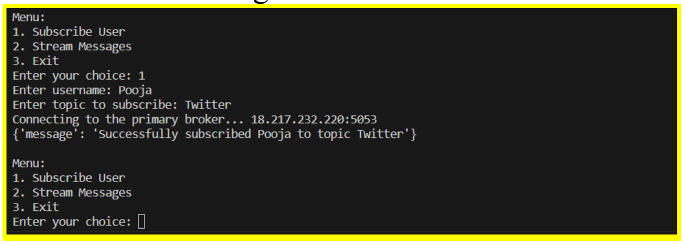
Test Case VII(Subscriber offline/Client Failure):

- Subscriber:
  - Username: Pooja
  - Topic : Twitter
- Publisher:
  - Topic : Twitter
    - Hi Tweet

Result: Subscriber is offline and comes up after let say one hour and tries to listen via GET stream request. Still able to receive messages on reconnection.

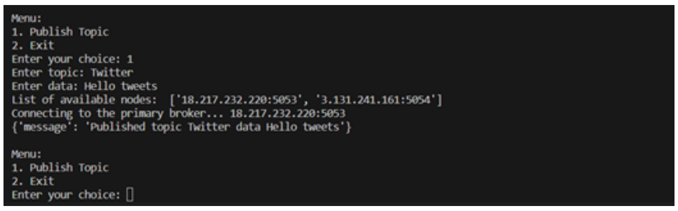
Snapshot:

Subscriber subscribed to the topic but has not streamed her messages:



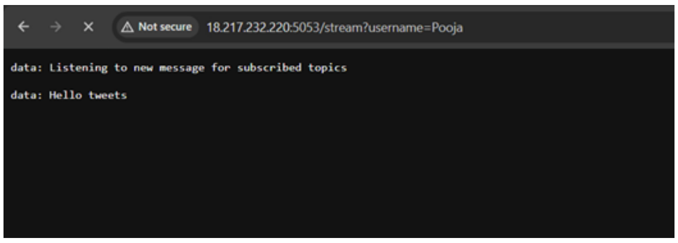
```
Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 1
Enter username: Pooja
Enter topic to subscribe: Twitter
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Successfully subscribed Pooja to topic Twitter'}

Menu:
1. Subscribe User
2. Stream Messages
3. Exit
Enter your choice: 
```

Publisher:



```
Menu:
1. Publish Topic
2. Exit
Enter your choice: 1
Enter topic: Twitter
Enter data: Hello tweets
List of available nodes:  ['18.217.232.220:5053', '3.131.241.161:5054']
Connecting to the primary broker... 18.217.232.220:5053
{'message': 'Published topic Twitter data Hello tweets'}

Menu:
1. Publish Topic
2. Exit
Enter your choice: 
```

Subscriber able to stream her messages when she is back online:



```
data: Listening to new message for subscribed topics

data: Hello tweets
```

Result and Summary of test cases implemented :
We have implemented well rounded scenarios to test our system.

- Our service is At least Once.
- We can see from test case 1 that multiple users subscribed to one same topic and multiple different topics are able to receive the messages.
- Also all the subscribers are getting only new message on stream request(no previous messages are get)
- For the system to be working primary or secondary at least one needs to be alive.
- Consistency between two brokers always remains intact as visible with continuing test cases from test case I to IV, where primary and secondary are leaving and rejoining the system.
- Test case V demonstrates that if both the system fails then also at restart they can restore the state by retrieving the permanent cache information which helps in restoring the brokers with previous transactions before their crash.
- Security of information is present as confirmed from test case VI
- Client failure is supported as confirmed with test case VII.

XI. Conclusions

This paper has presented a comprehensive study of a live news notification system based on the publisher-subscriber model, aimed at providing real-time updates within an organization. Throughout the project, several objectives were achieved, including the assurance of at least once message delivery, enhanced real-time communication, efficient handling of consistency, system reliability and availability, adaptability to organizational growth, and secure and controlled information flow.

The implementation of the system using Flask APIs, multi-threading, Heartbeat Protocol, and a dual-broker setup hosted on Amazon EC2 instances proved to be effective. The system showed remarkable fault tolerance, scalability, data integrity, consistency, security, concurrency, and performance. These features were validated through various test scenarios demonstrating the system's capability to handle different operational challenges such as broker failures, subscriber offline situations, and dynamic scaling.

1. Lessons Learned

Fault Tolerance and Reliability: The dual-broker setup was instrumental in achieving high availability. The use of Heartbeat and Replication and Consistency Protocols ensured continuous operation even during partial system failures.

Scalability and Performance: Using flask POST API to add new brokers and hosting the brokers on Amazon EC2 instances provided the necessary infrastructure to dynamically scale resources, accommodating increasing demands without compromising performance.

Security and Data Integrity: Restricting message receipt to registered subscribers and implementing robust data replication and consistency protocols safeguarded sensitive information and maintained data integrity.

2. Possible Improvements

Enhanced Load Balancing: Future iterations could incorporate more sophisticated load balancing mechanisms to distribute workloads more evenly across brokers by adding new brokers, especially under high-traffic conditions.

Improved Data Compression Techniques: Implementing advanced data compression techniques could optimize bandwidth usage and enhance system efficiency, particularly beneficial for organizations with large data flows.

AI-Based Predictive Analysis: Integrating AI algorithms for predictive analysis could enable the system to anticipate and prepare for potential system loads or failures, further improving reliability and performance.

Expanded Broker Network: Exploring the possibility of adding more brokers to the network could offer increased fault tolerance and scalability, allowing the system to cater to larger organizations or those with more complex needs.

User-Friendly Interface for Non-Technical Users: Enhancing the system's user interface to be more intuitive for non-technical users could improve the system's accessibility and ease of use.

In conclusion, this project has successfully demonstrated the viability and effectiveness of a pub-sub architecture for real-time organizational notifications. It offers a solid foundation for future advancements and adaptations to meet evolving organizational communication needs.

References

1. Donta, P. K., & Dustdar, S. (2023). Towards Intelligent Data Protocols for the Edge.
2. Matic, M., et al. (2020). Scheduling messages within MQTT shared subscription group in the clustered cloud architecture.
3. Longo, E., Redondi, A. E. C., Cesana, M., Arcia-Moret, A., & Manzoni, P. MQTT-ST: a Spanning Tree Protocol for Distributed MQTT Brokers.
4. Kreps, J.(2011). Kafka: a Distributed Messaging System for Log Processing.
5. Ghosh, H.Message Diffusion in Distributed Publish-Subscribe Systems.
6. Salehi, P., Zhang, K., & Jacobsen, H.-A. PopSub: Improving Resource Utilization in Distributed Content-based Publish/Subscribe Systems.