Distributed Emergency and Traffic Incident Notification System

Distributed Systems- COEN 317

Professor Ramin Moazzeni

Naveena Avula - 07700009231

Sri Sai Saketh Chillapalli - 007700000089

Karumanchi Samuel - 07700009611

Department of Computer Science and Engineering

School Of Engineering

Distributed Emergency and Traffic incident Notification System

Naveena Avula, Sri Sai Saketh Chillapalli and Karumanchi Samuel

COEN317 Distributed System- Professor Ramin Moazzeni

Santa Clara University

Abstract– "Distributed Emergency and Traffic incident Notification system" leverages a sophisticated publish/subscribe system, designed to provide real-time, personalized traffic incident or emergency notifications within a specific area. The system facilitates a topic-centric subscription model within a distributed network, enabling the efficient targeting of incidents and emergencies to subscribers based on their specific interests. Emphasizing decentralized processing, " Distributed Emergency and Traffic incident Notification system " relies on the RabbitMQ message broker and Kubernetes for orchestration, which collectively enhance the system's scalability and ensure the consistent delivery of messages. The platform adeptly informs people about the local traffic updates, allows users to subscribe to receive notifications based on their geographical area, types of emergencies they're interested in, and preferred notification methods.

I. MOTIVATION

In today's interconnected world, the rapid and accurate dissemination of emergency information and traffic incidents is crucial for public safety and efficient urban mobility. Current systems often suffer from limitations such as delayed notifications, irrelevant alerts, and lack of personalization, which can lead to confusion, inefficient resource allocation, and potentially life-threatening situations. The proposed Distributed Emergency and Traffic Incident Notification System aims to address these challenges by using distributed systems technology and a publisher-subscriber model.

II.DISTRIBUTED SYSTEMS CHALLENGES

The development and implementation of the *Distributed Emergency and Traffic Incident Notification System* introduce a robust distributed system aimed at enhancing communication and information dissemination in an urban environment. This system focuses on the coordination of emergency alerts, traffic incident updates, and user-specific notifications. While the project seeks to provide timely and relevant information to users, it must address and overcome the unique challenges associated with distributed systems.

A. Decentralized Communication

Urban areas are complex ecosystems with numerous events and incidents occurring simultaneously. Coordinating and delivering information across decentralized networks is challenging, particularly in maintaining reliable publisher-subscriber communication. The implemented model leverages decentralized communication to allow users to subscribe to specific alert types, ensuring the efficient and personalized delivery of critical information.

B. Information Overload

Traditional communication methods, such as mass alerts, often lead to information overload. Users in urban areas may receive notifications about incidents unrelated to their immediate surroundings, causing them to overlook critical updates. A publish-subscribe architecture, informed by research, allows users to customize their notification preferences. This personalized system reduces information overload, ensuring that users receive alerts relevant to their specific location and needs, enhancing the effectiveness of emergency and traffic incident notifications.

C. Scalability and Performance

In distributed notification systems, maintaining system efficiency and guaranteeing scalability become crucial concerns as the number of users and occurrences rises. In order to manage increasing requests and provide timely and pertinent notifications, the system architecture incorporates concepts from effective publish/subscribe models described in the literature.

D. Fault Tolerance and Reliability

To guarantee continuous service, distributed systems need to be resistant to errors and malfunctions. Users may miss important messages as a result of outages or interruptions. Using RabbitMQ and Kubernetes improves fault tolerance by enabling other instances to take over without interruption in the event of a failure. The system's reliability is greatly increased by this architecture, which guarantees constant delivery of updates on traffic incidents and emergencies.

E. Consistency and Event Ordering

Delivering a precise and reliable user experience requires maintaining consistency in the sequence of events and notifications. Even in high-concurrency situations, the system preserves consistency and orderly event processing by fusing the publish-subscribe pattern with mutual exclusion algorithms and strong concurrency management. Users are guaranteed dependable and well-structured updates with this method.

F. Integration with External APIs

Fetching and integrating data from external APIs, such as Disaster API, introduces challenges related to data synchronization and API reliability.The system incorporates error handling mechanisms to handle external API interactions, ensuring smooth integration and minimizing disruptions. In addressing these distributed system challenges, Distributed Emergency and Traffic Incident Notification System aims to provide a robust and user-friendly platform that effectively connects users with valuable opportunities while maintaining the reliability and scalability required in a university setting.

III. LITERATURE REVIEW

The Distributed Emergency and Traffic Incident Notification System, which utilizes a publish/subscribe (pub/sub) messaging framework, has been greatly informed by extensive research highlighted in various IEEE papers. The insights from these studies have played a pivotal role in shaping the design and implementation of the system's pub/sub architecture.

1. "A Dynamic Failure Detector for P2P Storage System" (IEEE, 2009) This paper discusses the challenges of failure detection in unreliable distributed systems. It proposes adaptive techniques which dynamically adjust to changing network conditions. We plan to adapt these concepts to ensure our system remains robust and can differentiate between genuine failures and temporary delays in emergency information propagation.

   Key takeaways:

   - Adaptive failure detection techniques
   - Balancing speed and accuracy in failure detection
   - Considerations for unreliable network conditions
2. "**Overlay routing network construction by introducing Super-Relay nodes**" (IEEE, 2014) This research presents an approach to optimize routing quality in resilient and scalable networks. The concept of Super-Relay nodes could be particularly useful in our system to ensure efficient delivery of emergency notifications across wide areas.

   Key takeaways:

   - Use of Super-Relay nodes for improved performance
   - Strategies for optimizing network topology
   - Balancing between routing efficiency and network resilience
3. "**SELECT: A Distributed Publish/Subscribe Notification System for Online Social Networks**" (IEEE, 2018) This paper proposes a pub/sub notification system using a P2P network with a ring topology. While our system isn't focused on social networks, their approach to reducing overhead by adapting connections based on user relationships could be adapted to our geographical and preference-based model.

   Key takeaways:

   - P2P network with ring topology for pub/sub systems
   - Techniques for reducing communication overhead
   - Adapting network connections based on user relationships (which we can apply to geographical proximity)
4. "**Research and design of Pub/Sub Communication Based on Subscription Aging**" (IEEE, 2018) This article introduces a subscription aging-based pub/sub communication model with prioritized subscribers and multicast transmission. These concepts could be valuable in our system, especially for prioritizing critical alerts and managing long-term subscriptions.

   Key takeaways:

   - Subscription aging mechanisms
   - Prioritization of subscribers in pub/sub systems
   - Efficient multicast transmission techniques
5. "**Improving the Performance of a Publish-Subscribe Message Broker**" (Rocha et al., 2019) This work explores techniques to optimize message brokers for IoT systems. Their approach to refining event handler mechanisms and optimizing communications could be directly applicable to reducing latency in our emergency notification system.

   Key takeaways:

   - Optimization techniques for pub/sub message brokers
   - Strategies for reducing communication latency
   - Scalability considerations for large-scale pub/sub systems.

6. "**EdgePub: A Dynamic Publish/Subscribe Framework for IoT at Edge**" (ACM, 2022) - This research presents an edge computing-based pub/sub system that significantly reduces latency in IoT environments. Their approach to handling mobile subscribers and dynamic topic management could be valuable for our location-based emergency notifications.

   Key takeaways:

   - Edge-based architecture for reduced latency
   - Dynamic topic management
   - Mobile subscriber handling techniques

These papers provide a solid foundation for our project, offering insights into key challenges and potential solutions in areas such as failure detection, efficient routing, security, and performance optimization. We will incorporate these learnings into our system design and implementation to ensure a robust, efficient, and secure emergency notification system.

IV. PROJECT DESIGN

A Minikube cluster hosts the RabbitMQ server, a vital part of the system, in the architecture seen in Fig. 1. A program called Minikube enables you to run Kubernetes locally, offering a productive method for testing and deploying apps in a containerised environment that resembles a production setting. In this case, RabbitMQ gains from Kubernetes' scalability and manageability, which strengthens its resilience and dependability as a message broker.

The producer is a Flask server that runs on localhost at port 5000 and is implemented in producer.py. This server is in charge of publishing and disseminating messages to the RabbitMQ server's many topics. Conversely, the consumer runs on localhost at port 5001 and is included in consumer.py.
It manages message retrieval from the RabbitMQ server as well as subscriptions and un - subscriptions.

The RabbitMQ server, which is housed on the Minikube cluster, may be reached by both the producer and consumer applications. Despite being in separate contexts, this configuration guarantees a smooth communication between the message broker and the Flask applications.

The Kubernetes service is used to expose RabbitMQ in order to enable this communication. With this method, the RabbitMQ server within the Minikube cluster can be accessed by external apps, such as our Flask servers running on localhost. By serving as a conduit between the Flask apps and RabbitMQ, the service makes sure that messages move freely throughout the pub/sub system.

This architecture, leveraging Minikube's capabilities, offers a robust, scalable, and flexible solution. It allows for a development and testing environment that closely mirrors production settings, aiding in smoother deployments and scalability testing.

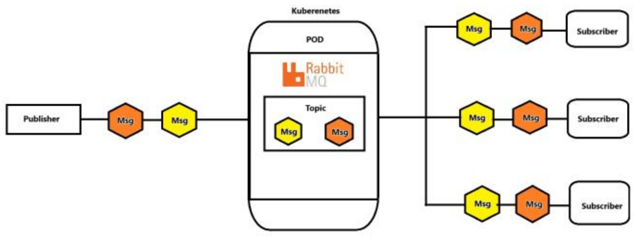


Fig. 1 High Level Architecture

V. EVALUATION

Using advanced algorithms for broadcasting, replication, concurrency, and mutual exclusion, the Distributed Emergency and Traffic Incident Notification System is a strong illustration of distributed messaging. These advanced methods preserve reliable and secure communication over a decentralized pub/sub network while improving fault tolerance, performance, and scalability.

A. Broadcast algorithm implementation

Broadcasting is the capability of sending a message to every subscriber in a pub/sub (publish/subscribe) system, independent of each subscriber's specific topic subscriptions. This is especially helpful for sharing important or urgent information that has to instantly reach a large audience, such alerts or notifications. Topic-based messaging was first supported by our system. We strategically changed the current infrastructure to provide this system broadcasting capabilities, enabling it to process broadcast messages in addition to standard topic messages.

'Broadcast' is the unique broadcast topic that we defined. All subscribers get broadcast messages using this special identification. The Flask producer application now has a new endpoint, /broadcast. Broadcast messages are accepted by this endpoint. The application uses the 'broadcast' routing key to publish a broadcast message to the RabbitMQ exchange after receiving it. The queue of each subscriber is set up to bind to both the broadcast topic and their own topic. Using the RabbitMQ queue_bind function, this is accomplished. To manage and distinguish between broadcast messages and standard topic-specific messages, the consumer logic has been modified. This guarantees that subscribers are able to comprehend and react to broadcast messages in a suitable manner.

Process Flow: The Flask application's /broadcast endpoint in producer.py receives an urgent message that has to be broadcast. With a broadcasting-specific routing key, the Flask application, in its capacity as the producer, publishes this message to the RabbitMQ exchange. After then, RabbitMQ sends this message to every queue associated with the broadcast routing key. All subscribers receive the broadcast message because each subscriber's queue is tied to this key in addition to their own topics. The message is retrieved and processed by each subscriber, also known as a consumer. The broadcast message is acknowledged and dealt with appropriately thanks to the consumer logic. We can avoid the hassle of administering several exchanges or changing the messaging system's fundamental architecture by utilising a unique broadcast topic within the same exchange. To receive broadcast messages, subscribers do not need to sign up for more subjects or exchanges.

B. Replication

A strong replication strategy is essential to the architecture of the Distributed Emergency and Traffic Incident Notification System, improving availability and fault tolerance. Kubernetes, which coordinates containerised instances of the RabbitMQ message broker throughout a cluster, is used to achieve this technique. By using Kubernetes Deployments to deploy numerous RabbitMQ replicas, as stated in a rabbitmq-deployment.yaml file, resilience is further strengthened and a predefined number of Pod replicas are constantly operational.

Kubernetes Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) are used to control state and message durability, protecting against data loss during Pod restarts or rescheduling. Kubernetes Services provide reliable connectivity to the RabbitMQ broker and facilitate effective traffic allocation between replicas. In order to provide system resilience, Kubernetes' health check probes continuously monitor the RabbitMQ Pods and allow self-healing by automatically restarting unresponsive Pods.

This thorough replication approach guarantees continuous processing by enabling the system to tolerate node failures and handle traffic spikes with ease. The deployment demonstrates how Kubernetes can offer a scalable and robust infrastructure for distributed applications, satisfying the high availability and dependability requirements necessary for traffic incident and emergency alerting systems.

C. Concurrency and Mutual Exclusion

The interaction between threads and locks is essential for controlling concurrency and guaranteeing the dependability of the Distributed Emergency and Traffic Incident Notification System. Because each consumer is given its own thread, a multi-threaded environment is created, allowing message processing to take place concurrently across several queues. By increasing the system's ability to manage large numbers of messages at once, this threading approach improves scalability and task allocation.

A new thread is started to handle messages for a topic when a user subscribes to it, enabling concurrent, non-blocking actions. Even when the number of users increases, the system's responsiveness and high throughput are guaranteed by the threads' independence.

The system uses locks to efficiently handle mutual exclusion in order to enable this threading design. Access to shared resources is synchronised by a crucial lock called subscribe_lock, especially when subscribing and unsubscribing. To avoid race situations and preserve data integrity, the lock protects these actions, which entail changing message queue states. To prevent crucial parts from running concurrently, the system makes sure that only one thread can bind or unbind queues to the exchange at a time.

The system's advanced concurrency management is demonstrated by the combination of locks for synchronised access and threads for concurrent message processing. The system ensures secure and reliable state transitions by requiring threads to acquire, execute, and release locks during sensitive operations. With the help of this dual approach, the notification system can effectively manage the concurrent and changing needs of its users, offering a message service that is incredibly dependable and efficient.

D. Efficiency and Resilience of the System

In order to provide dependable and effective operation in metropolitan areas, the Distributed Emergency and Traffic Incident Notification System addresses important distributed system challenges:

1) Fault Tolerance: The system makes use of RabbitMQ's message queuing features, which offer fault tolerance by using persistent exchange declarations and message persistence (delivery_mode=2). Container orchestration with self-healing capabilities, like automated restarts of failing containers, is made possible by Kubernetes, guaranteeing fault-free recovery. Because consumer threads have retry and connection error handling capabilities, they can continue to function even in the event of brief network outages.

2) Performance: RabbitMQ's sophisticated message routing and load distribution enable effective message handling. Processing HTTP requests quickly and with little overhead is guaranteed by the lightweight Flask framework. In order to process and promptly send vital traffic or emergency data to customers, the producer service effectively communicates with external APIs.

3) Scalability: By modifying the number of pod copies in response to system load, Kubernetes facilitates horizontal scalability of services, guaranteeing that the system grows to accommodate demand. Performance is maintained under heavy demand because to RabbitMQ's strong exchange and queue algorithms, which can handle growing numbers of users and messages.

4) Consistency: To assure that all updates spread throughout the network over time and that consumers receive correct and comprehensive data, the system uses eventual consistency. RabbitMQ makes sure that messages are reliably routed to the right queues by using binding keys and exchanges.

5) Concurrency: Threading is used in the consumer service to manage concurrency, allowing several consumers to process messages simultaneously without interfering. Race situations are avoided via thread-safe operations, which are secured by locks and guarantee the integrity of state-changing activities like subscribing and unsubscribing.

6) Security: While HTTPS encrypts contact with external services (for example Disaster API), protecting data in transit, HTTP Basic Authentication safeguards interactions with RabbitMQ's administration API.

Strong techniques to handle common distributed system issues, such as fault tolerance, performance, scalability, consistency, concurrency, and security are incorporated into the Distributed Emergency and Traffic Incident Notification System. When combined, these processes guarantee a robust, effective, and scalable system that offers consumers in metropolitan areas secure and dependable notifications.

VI. IMPLEMENTATION DETAILS

The foundation of the Distributed Emergency and Traffic Incident Notification System are producer.py and consumer.py, two essential Python Flask applications that communicate with a RabbitMQ server acting as the message broker. The publish/subscribe (pub/sub) paradigm, which is the best way to handle event-driven communication in distributed systems, is used in this architecture. Users in metropolitan areas will receive emergency notifications and traffic updates efficiently because of this design.

A. Flask for Application Development

A Python web framework known for its ease of use and efficiency, Flask is lightweight and adaptable. It is the perfect solution for creating small to medium-sized web services or APIs since, as a microframework, it requires very little boilerplate code.

Flask's extensibility is one of its best qualities. It can be extended with a variety of extensions for tasks like database integration, form validation, and user authentication, even though it offers the fundamental tools required to create online applications, such as managing responses and routing requests.

Applications are kept effective and customized to meet particular needs of developers using this modular approach.

Flask's design places a strong emphasis on minimalism and simplicity. It is a useful and dependable framework for application development because of its essential features, which include an integrated development server and debugger, support for unit testing, and RESTful request dispatching. Flask plays an essential role in the producer.py and internal.py components of the Distributed Emergency and Traffic Incident Notification System. The development of web services for publishing and subscribing to messages in a pub/sub architecture is made easier by its speedy establishment of HTTP endpoints. Flask's flexibility with complex distributed system designs and message brokers is demonstrated by its smooth integration with RabbitMQ.

B. The Publisher

By acting as the publisher, the producer.py program allows the system to broadcast messages on a range of subjects. This application, which was developed with Flask, leverages pika, a Python client library for RabbitMQ, to connect to RabbitMQ which is a powerful message broker. The connection is set up

with characteristics including host, port, credentials, and heartbeat settings to guarantee efficient communication while preserving stability and resilience.
The application configures the necessary elements for message routing after connecting to RabbitMQ. These consist of a designated routing key, the BROADCAST_ROUTING_KEY, and the ROUTING_EXCHANGE. When combined, these components allow for the effective delivery of traffic incident and emergency alert notifications to specified queues, guaranteeing that messages are received by the right users in the distributed system. This setup emphasizes how important the publisher is to the notification architecture.

*1) Exchange Name (*ROUTING_EXCHANGE*):* An exchange serves as a message routing agent in RabbitMQ, receiving in messages from publishers and distributing them to queues in accordance with predetermined guidelines. The exchange is of type 'topic' and is called 'routing' in the Distributed Emergency and Traffic Incident Notification System. By matching the routing key of the message with the binding patterns which connect queues to the exchange, this type enables messages to be routed to queues. This guarantees effective and precise notification delivery.

*2) Broadcast subject(*BROADCAST_ROUTING_KEY*):* The system has a specific subject called the "broadcast" topic that is used to broadcast messages to every subscriber. This topic is used for sending urgent or general alerts that must reach all connected users, regardless of their unique subscriptions. This feature guarantees that everyone receives important information, such as emergency alerts, as soon as possible.

3) Routing Key: An essential feature that the exchange uses to decide how to route messages to queues is the routing key. The 'topic' attribute of the incoming message determines the routing key's dynamic assignment in this application. Flexible and accurate message delivery is made possible by this dynamic assignment, which routes messages to the broadcast topic for broader distribution or to designated queues for specialized topics. This system guarantees that notifications reach their target customers in an efficient manner.

Additionally, a crucial function for obtaining data from other sources is retrieve_external_data (), which has been integrated in the program. In particular, the Disaster API endpoint that this function communicates with.

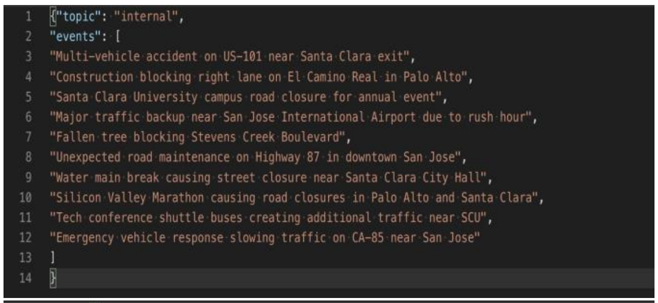
The function sends an HTTP GET request to the specified URL when it is called. It parses and provides the JSON data that was retrieved from the API after obtaining a successful response (HTTP status code 200). The 'external' topic in the system is then published messages using this retrieved data.

The function increases the system's capacity to provide important updates from external sources by permitting the integration of external data. This feature guarantees that the Distributed Emergency and Traffic Incident Notification System will continue to be adaptable and dynamic, able to broadcast and integrate important external information along with internal notifications.

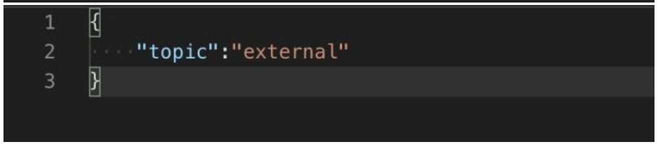
To make message publication and distribution easier, the application has two main endpoints: publish and broadcast.

1)publish Endpoint: Both internal and external communications are handled by this versatile endpoint.

- It publishes events to the designated internal topic after reading them from an API call from POSTMAN.
- Using the retrieve_external_data () function, it obtains information from the Disaster API for external subjects and publishes it to the external topic
- The topic given in the POST request is used by the endpoint to identify whether the data is internal or external



```
{"topic": "internal",
"events": [
"Multi-vehicle accident on US-101 near Santa Clara exit",
"Construction blocking right lane on El Camino Real in Palo Alto",
"Santa Clara University campus road closure for annual event",
"Major traffic backup near San Jose International Airport due to rush hour",
"Fallen tree blocking Stevens Creek Boulevard",
"Unexpected road maintenance on Highway 87 in downtown San Jose",
"Water main break causing street closure near Santa Clara City Hall",
"Silicon Valley Marathon causing road closures in Palo Alto and Santa Clara",
"Tech conference shuttle buses creating additional traffic near SCU",
"Emergency vehicle response slowing traffic on CA-85 near San Jose"
]
}
```

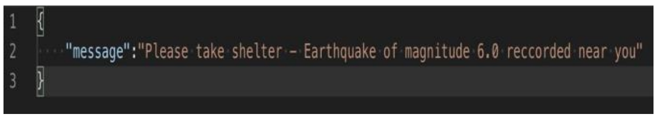


```
{
    "topic":"external"
}
```

Two key-value pairs are included in the JSON-formatted payload for POST requests:

- message: Information that will be delivered
- topic: The kind of topic (for example, "external" or "internal")

A simple and structured method of sending data to the appropriate HTTP endpoints is ensured by this JSON-based format. It makes it easier to process messages efficiently and route them accurately inside the pub/sub system by clearly identifying the topic and content of the messages. This architecture keeps the notification distribution process flexible and dependable while facilitating the smooth integration of internal and external data sources.



```
{
    "message":"Please take shelter - Earthquake of magnitude 6.0 reccorded near you"
}
```

The broadcast endpoint, on the other hand, is intended for urgent messages that must be widely shared using a predetermined broadcast topic.

C. The Subscriber - Consumer.py

producer.py can be enhanced with the consumer.py program, which manages message retrieval and subscriptions. By connecting to RabbitMQ, this Flask-based application allows users to listen for messages sent to particular topic. /subscribe and /unsubscribe are its two main endpoints.

Users or services can subscribe to particular topic using the /subscribe endpoint. Upon a user's subscription:

- A distinct queue is made and given the user's name.
- By binding the queue to the designated topic, the user is guaranteed to get messages that are relevant to that subject.
- The user can receive both targeted and general broadcast messages because the queue is also tied to the broadcast topic.

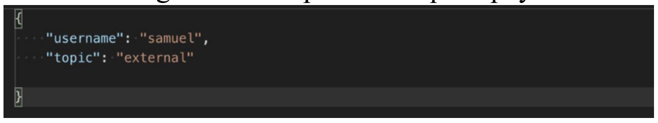
/unsubscribe Endpoint: The user's subscription to a certain topic is terminated using this endpoint by

- Removing the connection between the designated topic and the user's queue.
- Ensuring that messages about that subject are no longer sent to the user.

Two key-value pairs make up the JSON-formatted payload for POST requests directed at these endpoints

- The username of the person submitting the request is identified.
- Topic: Indicates the subject of the subscription or unsubscription.

The following is an example of a request payload:



```
{
    "username": "samuel",
    "topic": "external"

}
```

D. The role RabbitMQ Plays

As a reliable and effective message broker, RabbitMQ forms the foundation of the Distributed Emergency and Traffic Incident Notification System. Based on the designated aspects, it is in charge of directing publisher messages to the relevant subscriber queues. RabbitMQ is essential to this architecture because of its key functionalities, which include:

- Topic-based Routing: By allowing dynamic and adaptable message routing through RabbitMQ's topic exchange type, subscribers can still get broadcast messages while only receiving messages that are related to their interests.
- Scalability: RabbitMQ is well-suited to the requirements of a distributed pub/sub system because it is built to manage large message volumes.
- Reliable Delivery: RabbitMQ guarantees that messages are delivered consistently even in the case of brief interruptions or errors by enabling message persistence and acknowledgments.

The reliability of RabbitMQ is essential to this system because it guarantees that traffic notifications and emergency warnings are sent to the right people in a timely and correct manner, preserving the communication network's efficacy and integrity.

E. Sequence Diagram

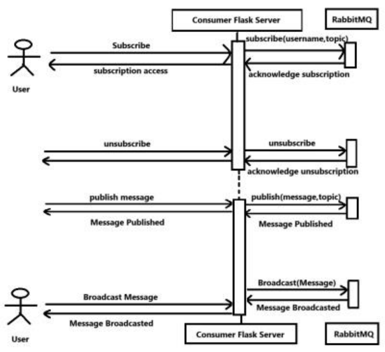


Fig. 2. Sequence Diagram

The Distributed Emergency and Traffic Incident Notification System's interactions are depicted in the sequence diagram in Figure 2, which emphasizes the communication between users, the Producer Flask server, the Consumer Flask server, and RabbitMQ. The flow of user-initiated actions, including publishing, broadcasting, or subscribing and how these actions are handled by different system components are graphically depicted in the diagram. It offers a brief overview of the dynamic interactions that make it possible for notifications to be delivered throughout the system with ease.

The process starts with a user submitting a subscription request to the Consumer Flask server, as shown in Fig. 2. In order to register the subscription and create or update the required queue bindings for the given topic, the server responds to this request by interacting with RabbitMQ. Following a successful registration, the user receives an acknowledgment from the system verifying their subscription.

In a similar manner, a user's request to the Consumer Flask server starts the unsubscription procedure. To remove the queue binding for the designated subject, the server talks to RabbitMQ. An acknowledgment is

provided back to the user to confirm that the unsubscription process was successful.
Users can effectively customize their notifications inside the Distributed Emergency and Traffic Incident system through these interactions, which provide smooth subscription and unsubscription administration.

In addition, Fig. 2 shows how a message is published, beginning with the user submitting a request to the Producer Flask server. After handling the request, the server talks to RabbitMQ to publish the message to the designated topic. An acknowledgment verifying the action is given back to the user when the message has been successfully published.
The diagram also shows how a message is broadcast, with the Producer Flask server sending it to a RabbitMQ broadcast topic that has been predefined. After that, RabbitMQ spreads the word to every subscriber, irrespective of their specific topic subscriptions. This demonstrates how the system can effectively and reliably handle a variety of notification needs by managing both broad communication for urgent notifications and focused messaging for certain topics.

F. Class Diagram

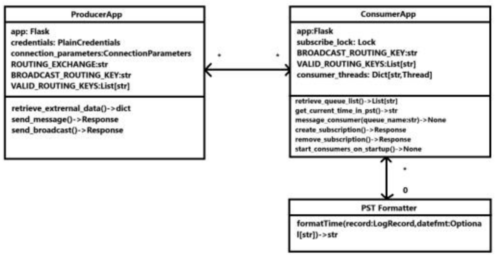

ProducerApp
app: Flask
credentials: PlainCredentials
connection_parameters:ConnectionParameters
ROUTING_EXCHANGE:str
BROADCAST_ROUTING_KEY:str
VALID_ROUTING_KEYS:List[str]
retrieve_extrernal_data()->dict
send_message()->Response
send_broadcast()->Response
*
*
ConsumerApp
app:Flask
subscribe_lock: Lock
BROADCAST_ROUTING_KEY:str
VALID_ROUTING_KEYS:List[str]
consumer_threads: Dict[str,Thread]
retrieve_queue_list()->List[str]
get_current_time_in_pst()->str
message_consumer(queue_name:str)->None
create_subscription()->Response
remove_subscription()->Response
start_consumers_on_startup()->None
*
0
PST Formatter
formatTime(record:LogRecord,datefmt:Optional[str])->str

Fig. 3. Class Diagram

The Distributed Emergency and Traffic Incident Notification System's object-oriented class diagram in Figure 3 shows the backend components' structural organization. It records the connections and interfaces between the ProducerApp and ConsumerApp classes, which work together to control the messaging system's vital functionality.

The functionality of the producer is represented by the ProducerApp Class. Important characteristics and techniques include:

- An instance of a Flask application for HTTP endpoint management.
- Connection parameters and RabbitMQ credentials to enable dependable communication with the message broker.
- Techniques for publishing messages, disseminating notifications to every subscriber, and retrieving external data.
- A list of permitted topics that is kept up to date to guarantee that communications follow the system's predetermined themes and stop unauthorized topic usage.

The consumer's operations are managed by the ConsumerApp Class. Important characteristics include:

- To guarantee thread safety when subscribing and unsubscribing, a subscription lock is used.
- A dictionary of consumer threads that allows messages to be processed in parallel for increased responsiveness and scalability.
- Techniques for handling subscriptions and unsubscriptions, retrieving queues, and starting consumer threads at system startup.

In order to provide effective message publishing, consumption, and user management within a strong distributed architecture, the ProducerApp and ConsumerApp classes work together to encapsulate the system's key processes.

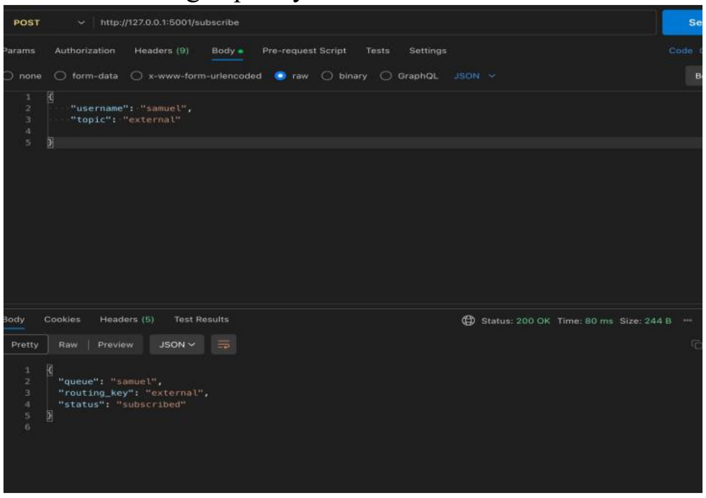
An essential tool for comprehending the backend architecture of the system is the class diagram shown in Figure 3. It displays a design that supports the pub/sub communication model by emphasizing efficiency, modularity, and process encapsulation.

VII. DEMONSTRATION OFSYSTEMRUN

The producer service effectively broadcasts external notifications to specified users after retrieving external data, including job ads, via the Disaster API (Fig. 8). In a similar manner, APIs are used to handle and broadcast internal events (Figs. 9 and 10). Message dependability is ensured via persistence and durability, which are guaranteed by both internal and external events (Fig. 14).

All active consumers receive messages from the broadcast functionality, which functions as planned (Fig. 12). The customer service has the ability to initialise consumer threads at startup and dynamically maintains user subscriptions. Race conditions are avoided even during concurrent requests because to the efficient maintenance of synchronisation during subscription and unsubscription processes.
The Distributed Emergency and Traffic Incident Notification System's functionality as a fully functional messaging system is confirmed by its demonstration in a Minikube-like environment (Fig. 13). The system's comprehensive design for publishing and receiving messages validates its deployment readiness and allows it to handle real-world circumstances. The system's capacity to reliably manage and route messages is demonstrated by the

successful run, underscoring Kubernetes's potency in orchestrating distributed systems and guaranteeing users receive high-quality notifications.



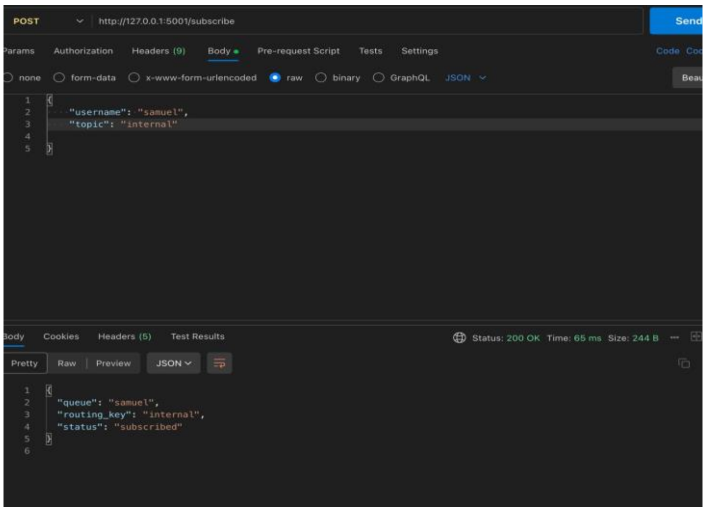
```
POST http://127.0.0.1:5001/subscribe
```

```
{
    "username": "samuel",
    "topic": "external"
}
```

Status: 200 OK Time: 80 ms Size: 244 B

```
{
    "queue": "samuel",
    "routing_key": "external",
    "status": "subscribed"
}
```

Fig. 4 Subscribing to external events



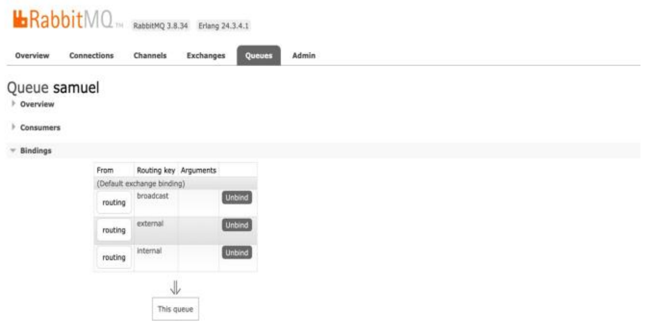
```
POST http://127.0.0.1:5001/subscribe
```

```
{
    "username": "samuel",
    "topic": "internal"
}
```

Status: 200 OK Time: 65 ms Size: 244 B

```
{
    "queue": "samuel",
    "routing_key": "internal",
    "status": "subscribed"
}
```

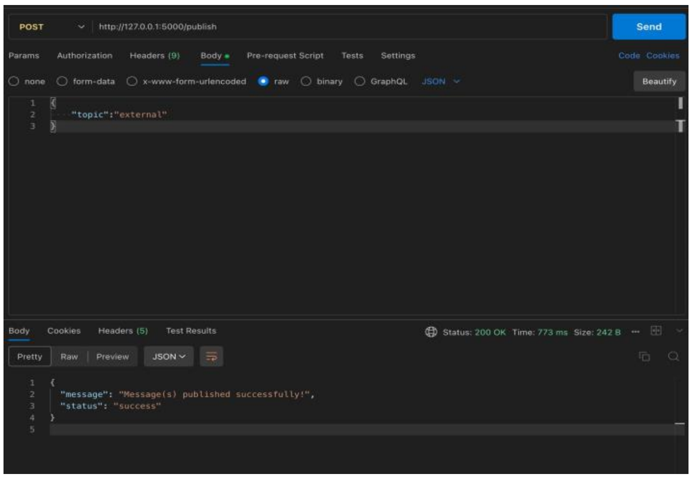
Fig. 5 Subscribing to internal events



Queue samuel

From	Routing key	Arguments
(Default exchange binding)		
routing	broadcast	
routing	external	
routing	internal	

Fig. 6 Queues Created in RabbitMQ UI



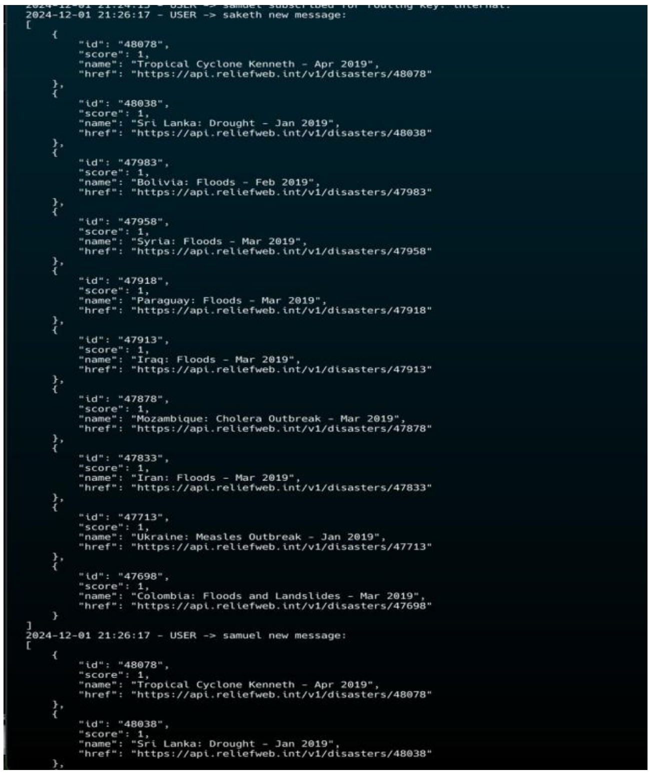
```
POST http://127.0.0.1:5000/publish
```

```
{
    "topic":"external"
}
```

Status: 200 OK Time: 773 ms Size: 242 B

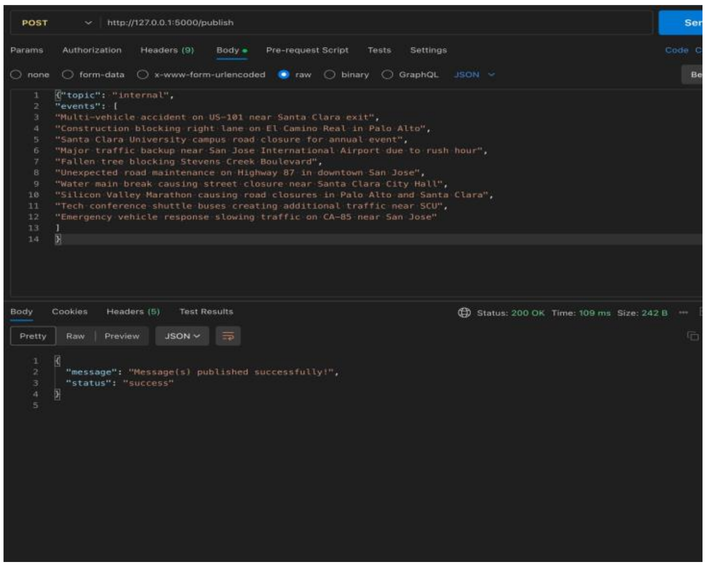
```
{
    "message": "Message(s) published successfully!",
    "status": "success"
}
```

Fig. 7 Publishing External Events



```
2024-12-01 21:24:15 - USER -> samuel subscribed for routing key: internal.
2024-12-01 21:26:17 - USER -> saketh new message:
[
    {
        "id": "48078",
        "score": 1,
        "name": "Tropical Cyclone Kenneth - Apr 2019",
        "href": "https://api.reliefweb.int/v1/disasters/48078"
    },
    {
        "id": "48038",
        "score": 1,
        "name": "Sri Lanka: Drought - Jan 2019",
        "href": "https://api.reliefweb.int/v1/disasters/48038"
    },
    {
        "id": "47983",
        "score": 1,
        "name": "Bolivia: Floods - Feb 2019",
        "href": "https://api.reliefweb.int/v1/disasters/47983"
    },
    {
        "id": "47958",
        "score": 1,
        "name": "Syria: Floods - Mar 2019",
        "href": "https://api.reliefweb.int/v1/disasters/47958"
    },
    {
        "id": "47918",
        "score": 1,
        "name": "Paraguay: Floods - Mar 2019",
        "href": "https://api.reliefweb.int/v1/disasters/47918"
    },
    {
        "id": "47913",
        "score": 1,
        "name": "Iraq: Floods - Mar 2019",
        "href": "https://api.reliefweb.int/v1/disasters/47913"
    },
    {
        "id": "47878",
        "score": 1,
        "name": "Mozambique: Cholera Outbreak - Mar 2019",
        "href": "https://api.reliefweb.int/v1/disasters/47878"
    },
    {
        "id": "47833",
        "score": 1,
        "name": "Iran: Floods - Mar 2019",
        "href": "https://api.reliefweb.int/v1/disasters/47833"
    },
    {
        "id": "47713",
        "score": 1,
        "name": "Ukraine: Measles Outbreak - Jan 2019",
        "href": "https://api.reliefweb.int/v1/disasters/47713"
    },
    {
        "id": "47698",
        "score": 1,
        "name": "Colombia: Floods and Landslides - Mar 2019",
        "href": "https://api.reliefweb.int/v1/disasters/47698"
    }
]
2024-12-01 21:26:17 - USER -> samuel new message:
[
    {
        "id": "48078",
        "score": 1,
        "name": "Tropical Cyclone Kenneth - Apr 2019",
        "href": "https://api.reliefweb.int/v1/disasters/48078"
    },
    {
        "id": "48038",
        "score": 1,
        "name": "Sri Lanka: Drought - Jan 2019",
        "href": "https://api.reliefweb.int/v1/disasters/48038"
    },
```

Fig. 8 Displaying External Events


```
POST  http://127.0.0.1:5000/publish
```

```
{"topic": "internal",
"events": [
"Multi-vehicle accident on US-101 near Santa Clara exit",
"Construction blocking right lane on El Camino Real in Palo Alto",
"Santa Clara University campus road closure for annual event",
"Major traffic backup near San Jose International Airport due to rush hour",
"Fallen tree blocking Stevens Creek Boulevard",
"Unexpected road maintenance on Highway 87 in downtown San Jose",
"Water main break causing street closure near Santa Clara City Hall",
"Silicon Valley Marathon causing road closures in Palo Alto and Santa Clara",
"Tech conference shuttle buses creating additional traffic near SCU",
"Emergency vehicle response slowing traffic on CA-85 near San Jose"
]
}
```

Status: 200 OK Time: 109 ms Size: 242 B

```
{
  "message": "Message(s) published successfully!",
  "status": "success"
}
```

Fig. 9 Publishing Internal Events

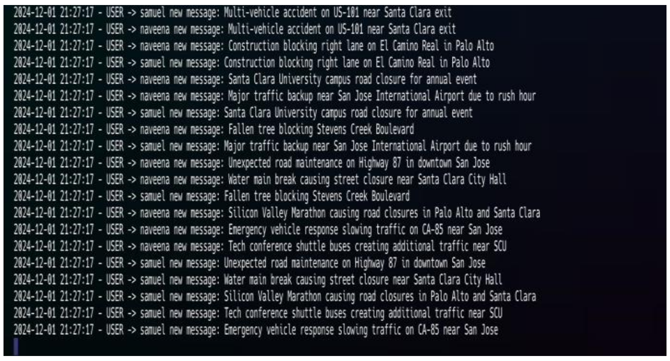
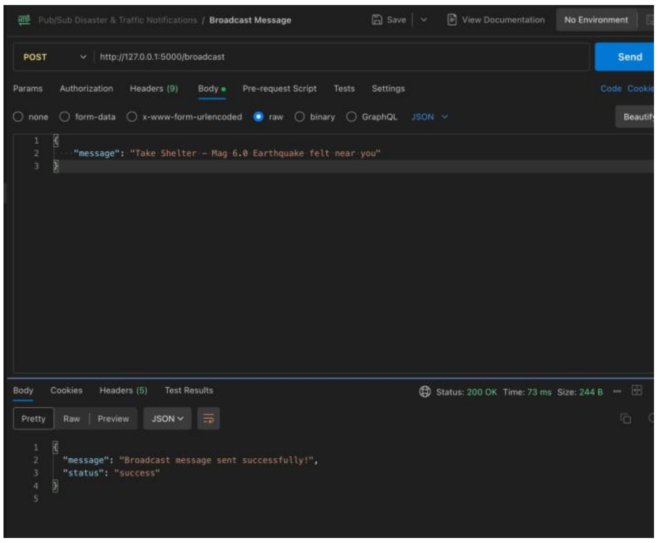

```
2024-12-01 21:27:17 - USER -> samuel new message: Multi-vehicle accident on US-101 near Santa Clara exit
2024-12-01 21:27:17 - USER -> naveena new message: Multi-vehicle accident on US-101 near Santa Clara exit
2024-12-01 21:27:17 - USER -> naveena new message: Construction blocking right lane on El Camino Real in Palo Alto
2024-12-01 21:27:17 - USER -> samuel new message: Construction blocking right lane on El Camino Real in Palo Alto
2024-12-01 21:27:17 - USER -> naveena new message: Santa Clara University campus road closure for annual event
2024-12-01 21:27:17 - USER -> naveena new message: Major traffic backup near San Jose International Airport due to rush hour
2024-12-01 21:27:17 - USER -> samuel new message: Santa Clara University campus road closure for annual event
2024-12-01 21:27:17 - USER -> naveena new message: Fallen tree blocking Stevens Creek Boulevard
2024-12-01 21:27:17 - USER -> samuel new message: Major traffic backup near San Jose International Airport due to rush hour
2024-12-01 21:27:17 - USER -> naveena new message: Unexpected road maintenance on Highway 87 in downtown San Jose
2024-12-01 21:27:17 - USER -> naveena new message: Water main break causing street closure near Santa Clara City Hall
2024-12-01 21:27:17 - USER -> samuel new message: Fallen tree blocking Stevens Creek Boulevard
2024-12-01 21:27:17 - USER -> naveena new message: Silicon Valley Marathon causing road closures in Palo Alto and Santa Clara
2024-12-01 21:27:17 - USER -> naveena new message: Emergency vehicle response slowing traffic on CA-85 near San Jose
2024-12-01 21:27:17 - USER -> naveena new message: Tech conference shuttle buses creating additional traffic near SCU
2024-12-01 21:27:17 - USER -> samuel new message: Unexpected road maintenance on Highway 87 in downtown San Jose
2024-12-01 21:27:17 - USER -> samuel new message: Water main break causing street closure near Santa Clara City Hall
2024-12-01 21:27:17 - USER -> samuel new message: Silicon Valley Marathon causing road closures in Palo Alto and Santa Clara
2024-12-01 21:27:17 - USER -> samuel new message: Tech conference shuttle buses creating additional traffic near SCU
2024-12-01 21:27:17 - USER -> samuel new message: Emergency vehicle response slowing traffic on CA-85 near San Jose
```

Fig. 10 Displaying Internal Events


Pub/Sub Disaster & Traffic Notifications / Broadcast Message
Save
View Documentation
No Environment

```
POST  http://127.0.0.1:5000/broadcast
```

```
{
    "message": "Take Shelter - Mag 6.0 Earthquake felt near you"
}
```

Status: 200 OK Time: 73 ms Size: 244 B

```
{
  "message": "Broadcast message sent successfully!",
  "status": "success"
}
```

Fig. 11 Broadcasting events

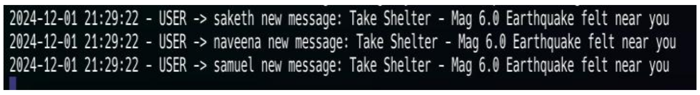
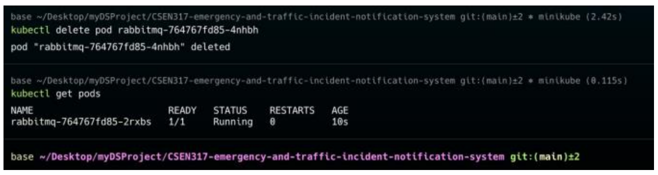

```
2024-12-01 21:29:22 - USER -> saketh new message: Take Shelter - Mag 6.0 Earthquake felt near you
2024-12-01 21:29:22 - USER -> naveena new message: Take Shelter - Mag 6.0 Earthquake felt near you
2024-12-01 21:29:22 - USER -> samuel new message: Take Shelter - Mag 6.0 Earthquake felt near you
```

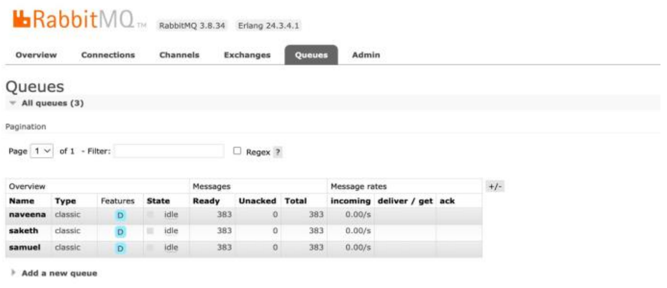
Fig. 12 Displaying broadcasted events


```
base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system git:(main)±2 ∗ minikube (2.42s)
kubectl delete pod rabbitmq-764767fd85-4nhbh
pod "rabbitmq-764767fd85-4nhbh" deleted

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system git:(main)±2 ∗ minikube (0.115s)
kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
rabbitmq-764767fd85-2rxbs   1/1     Running   0          10s

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system git:(main)±2
```

Fig. 13 Deleting Pod



Queues

All queues (3)

Pagination

Page 1 of 1 - Filter: Regex ?

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
naveena	classic	D	idle	383	0	383	0.00/s		
saketh	classic	D	idle	383	0	383	0.00/s		
samuel	classic	D	idle	383	0	383	0.00/s		

Add a new queue

Fig. 14 Messages persisted after POD Restart

VIII. PERFORMANCE ANALYSIS

Locust is a Python-based open-source load testing tool. It is employed to evaluate the effectiveness of APIs and web apps under different load scenarios. Locust's architecture is distinctive since it enables the creation of test scenarios in Python, providing a great deal of freedom and the capacity to script intricate user actions. It allows you to see how your system responds to increasing traffic and spot bottlenecks by simulating users accessing it. By simulating millions of users at once, it can shed light on how a system responds to high concurrency. Offers a web interface for real-time test progress monitoring.

The code locustfile.py uses Locust to simulate load on a pub/sub messaging system. The script is divided into two parts, each representing a different type of user interaction with the system: ProducerUser and ConsumerUser.

A. ProducerUser Class

It simulates the behavior of a producer in the pub/sub system.

1) *publish_internal Task:* This task posts a message to the /publish endpoint with the topic set to "internal". It simulates the action of publishing internal events.

2) *publish_external Task:* Similar to publish_internal, but for external events. It posts to the /publish endpoint with the topic "external".

3) *broadcast_message Task:* This task uses the /broadcast endpoint to simulate broadcasting an urgent message to all subscribers

B. ConsumerUser Class

It simulates the behavior of a consumer in the pub/sub system.

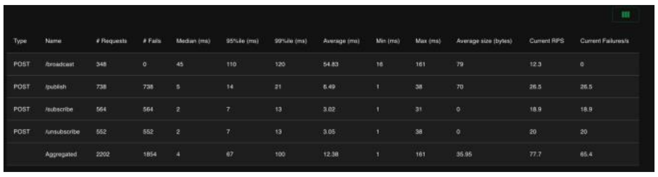
1) *subscribe_topic Task*: Sends a request to a hypothetical /subscribe endpoint, simulating a user subscribing to a specific topic.

2) *unsubscribe_topic Task:* Sends a request to an /unsubscribe endpoint, simulating a user unsubscribing from a specific topic.

A host and wait time are assigned to each user class. The wait time is the interval between each task execution, and the host is the base URL of the application under test. Locust generates instances of ProducerUser and ConsumerUser when the script executes.

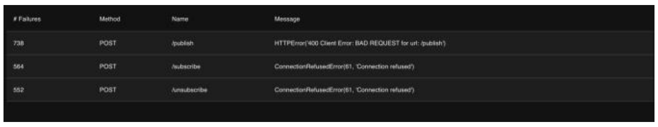
These instances carry out their duties by sending HTTP requests to the specified destinations. A comprehensive picture of the system's performance

under load can be obtained by examining the number of requests per second, response times, and failures using Locust's real-time monitoring interface. We can accurately model different user interactions with your pub/sub system using Locust, which provides us with important information about its performance features. This script specifically helps in understanding how the system scales with an increasing number of messages being published and how efficiently it handles subscriptions and broadcasts under load. It's a crucial part of ensuring that our system is robust and capable of handling real-world usage scenarios.
In particular, this script aids in comprehending how the system grows as more messages are released and how well it manages subscriptions and broadcasts when under strain. It's essential to making sure our system is reliable and able to manage usage scenarios found in the real world.



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/broadcast	348	0	45	110	120	54.83	16	161	79	12.3	0
POST	/publish	738	738	5	14	21	6.49	1	38	70	26.5	26.5
POST	/subscribe	564	564	2	7	13	3.02	1	31	0	18.9	18.9
POST	/unsubscribe	552	552	2	7	13	3.05	1	38	0	20	20
	Aggregated	2202	1854	4	67	100	12.38	1	161	35.95	77.7	65.4

Fig. 15 Request and Response time Statistics



# Failures	Method	Name	Message
738	POST	/publish	HTTPError('400 Client Error: BAD REQUEST for url: /publish')
564	POST	/subscribe	ConnectionRefusedError(61, 'Connection refused')
552	POST	/unsubscribe	ConnectionRefusedError(61, 'Connection refused')

Fig. 16 Failure statistics

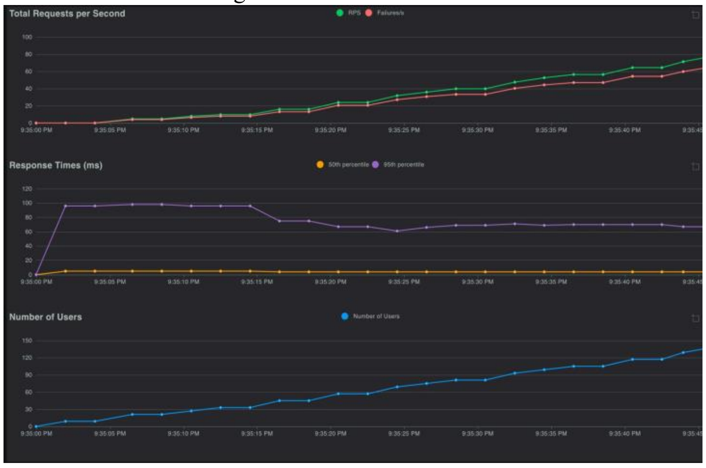


Fig. 17 Total requests per second, Response times and Number of Users

According to the Locust performance test report, Fig. 15 shows that the system evaluated a variety of API endpoints using HTTP POST techniques. There was only one unsuccessful request from the /publish endpoint (Fig. 16), but none from the others. 1,947 milliseconds was the average reaction time for all endpoints (Fig. 15).
Response times varied (Fig. 17), with 50% of requests on the /subscribe endpoint responding within 3,900 milliseconds, and the /unsubscribe endpoint responding within 3,700 milliseconds at the same percentile. However, the maximum response times for these endpoints were notably high at 10,244 and 12,826 milliseconds respectively, indicating potential bottlenecks or issues under high load. The system scaled up to 200 users (Fig. 17) over the test duration, with the number of requests per second peaking at 66.3. The user load was evenly split between producer and consumer users, with the former engaging in publishing (both internal and external) and broadcasting, while the latter subscribed and unsubscribed to topics.
In conclusion, the test performed well for the most part, with only one endpoint or publication displaying a failure, indicating a particular area that would require attention for enhancement. With faster response times in the 95th and 99th percentiles, the percentile breakdown of response times highlights the system's performance limits and offers insights into how the system behaves under pressure.

IX. TESTING RESULTS

The Distributed Emergency and Traffic Incident Notification System has undergone comprehensive unit testing to validate the functionality of its producer and consumer services. These tests were designed to verify that individual components of the system perform as intended in isolation. Below is a summary of the unit testing results and the planned approach for user testing of critical API endpoints:

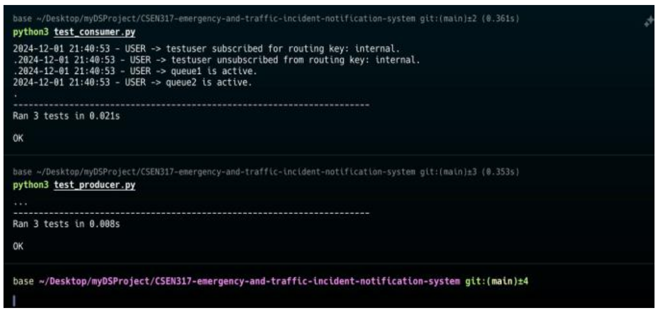
A. Unit Testing Results

1) Producer Service Tests

Test Broadcast Success: The broadcast functionality was tested to confirm that messages could be sent to all users successfully.
Test Fetch External Data Success: The system's ability to fetch external data from the Disaster API was validated.
Test Publish Internal Topic Success: The publishing service for internal topics was verified to ensure proper message dissemination within the system.
Outcome: As shown in Fig. 18 all tests are passed, indicating that the producer service is functioning correctly.

2) Consumer Service Tests:

Test Start Consumers on Startup: This test verified that when the system starts up, consumers (queues) are operational.
Test Subscribe Success: To make sure customers could successfully subscribe to topics, the subscription service was put through testing.
Test Unsubscribe Success: To make sure users could stop receiving messages from particular topics, the ability to unsubscribe from topics was verified.
Result: All tests pass, as illustrated in Fig. 18, indicating that the customer service is functioning as planned.



```
base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system git:(main)±2 (0.361s)
python3 test_consumer.py
2024-12-01 21:40:53 - USER -> testuser subscribed for routing key: internal.
.2024-12-01 21:40:53 - USER -> testuser unsubscribed from routing key: internal.
.2024-12-01 21:40:53 - USER -> queue1 is active.
2024-12-01 21:40:53 - USER -> queue2 is active.
.
----------------------------------------------------------------------
Ran 3 tests in 0.021s

OK

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system git:(main)±3 (0.353s)
python3 test_producer.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.008s

OK

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system git:(main)±4
```

Fig. 18 Unit test results for producer and consumer

B. User Testing Approach for API Endpoints

For comprehensive user testing of the system's API endpoints, the following methods will be applied:

1) Producer External (POST /publish): To make sure the system correctly retrieves and distributes job listings, user testing will mimic publishing events retrieved from the external Disaster API.

To verify the system's resistance to changes in external data, test cases will use a range of response situations from the Disaster API.
2) Producer Internal (POST /publish): To make sure the system processes and routes a number of internal messages appropriately, the internal publishing endpoint will be tested using a number of events. Checks for message persistence and delivery order will be included of the simulations.
3) Broadcast (POST /broadcast): Verifications on both the producer and consumer sides will guarantee that broadcast messages are received by all active users. Stress testing may be performed to evaluate the system's performance under high load.

4) Consumer Subscribe (POST /subscribe): Subscribing to internal and external topics will be part of user testing, and successful queue bindings and message receipts will be checked. Additionally, edge circumstances like subscribing to subjects that don't exist or using bogus usernames will be assessed.

5) Consumer Unsubscribe (POST /unsubscribe) API is to verify that unsubscribing successfully stops messages from being delivered to the user's queue.

To make sure the Un subscription procedure doesn't unintentionally impact other users or queues, more tests will be conducted.

The Distributed Emergency and Traffic Incident Notification System's various parts all work as anticipated, according to the unit tests. A dependable and effective platform for distributing important warnings will be ensured by the subsequent stage of user testing for the system's API endpoints, which will further validate its functionality in practical situations. The system exhibits its capacity to reliably handle and route messages while preserving a high level of service quality for its consumers thanks to a testing framework with extensive coverage.

X. CONCLUSION AND REFLECTIONS

The benefits and real-world implementation of the publish/subscribe paradigm in distributed systems are best demonstrated by the Distributed Emergency and Traffic Incident Notification System. Its creation required negotiating the intricacies of distributed systems, where important aspects like scalability and topic discovery were painstakingly planned and executed via in-depth investigation and the incorporation of cutting-edge protocols and algorithms.

The usage of Kubernetes for orchestration, which offers scalability and self-healing capabilities, and RabbitMQ for message queuing, which guarantees dependable and fault-tolerant communication, are essential components of the system's resilience. The design makes use of a replication approach to guarantee data availability and integrity as well as a broadcast algorithm for effective message distribution.

Threading and locking technologies are used to expertly manage concurrency and mutual exclusion within the customer service. By preventing race circumstances and enabling thread-safe operations, these safeguards guarantee the system's dependability in situations involving high concurrency.

The Distributed Emergency and Traffic Incident Notification System is essentially a scalable, robust solution that embodies fundamental distributed system concepts, going beyond a simple alerting platform. It serves as a paradigm for practical applications in the fields of traffic control and emergency communication due to its strong concurrency management, fault tolerance, and fast performance.

XI. REFERENCES

1. *S. Jiang, X. Song, H. Wang, J. -J. Han and Q. -H. Li, "A clustering-based method for unsupervised intrusion detections," Pattern Recognition Letters, vol. 27, no. 7, pp. 802-810, May 2006*
2. *Shengwen Tian, J. Liao, J. Wang and Qi Qi, "Overlay routing network construction by introducing Super-Relay nodes," 2014 IEEE Symposium on Computers and Communications (ISCC), Funchal, Portugal, 2014*
3. *N. Apolónia, S. Antaris, S. Girdzijauskas, G. Pallis and M. Dikaiakos, "SELECT: A Distributed Publish/Subscribe Notification System for Online Social Networks," 2018*

IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 2018

4. *D. Yang, M. Lian, Z. Zhang and M. Li, "The research and design of Pub/Sub Communication Based on Subscription Aging," 2018 IEEE International Conference on Intelligence and Safety for Robotics (ISR), Shenyang, China, 2018*
5. *R. Rocha, L. L. Ferreira, C. Maia, P. Souto and P. Varga, "Improving the performance of a Publish-Subscribe message broker," 2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC), Valencia, Spain, 2019*
6. *Pham*, V.-N.; Lee, G.-W.; Nguyen, V.; Huh, E.-N. Efficient Solution for Large-Scale IoT Applications with Proactive Edge-Cloud Publish/Subscribe Brokers Clustering. Sensors 2021.

XII. DIVISION OF WORK

Task	**Ownership**
RabbitMQ setup in Kubernetes, Consumer implementation	Samuel, Saketh
Producer Implementation, Broadcast Algorithm implementation	Saketh, Naveena
Replication, Concurrency and mutual exclusion algorithms implementation	Naveena, Samuel
Unit testing and Load testing	Team
Iteration-Implementing necessary changes	Team
Presentation Preparation, Code Submission and Final report preparation	Team