BRONCO CAREER ALERTS: PUB/SUB SYSTEM

Distributed Systems - COEN 317

Professor Ramin Moazzeni

Chaitanya Kothapalli(W1653623)- ckothapalli@scu.edu
Sri Lasya Malladi (W1650491)- smalladi2@scu.edu
Venkata Sai Srikar Jilla(W1652687) - vjilla@scu.edu

Github : https://github.com/jvsaisrikar/COEN317-BroncoJobAlerts

Department of Computer Science and Engineering

School Of Engineering

BRONCO CAREER ALERTS: PUB/SUB SYSTEM

Chaitanya Kothapalli, Sri Lasya Malladi, Venkata Sai Srikar Jilla

COEN 317 Distributed System - Professor Ramin Moazzeni

Santa Clara University

Abstract **– "Bronco Career Alerts" leverages a sophisticated publish/subscribe system, designed to provide real-time, personalized career notifications within a student community. The system facilitates a topic-centric subscription model within a distributed network, enabling the efficient targeting of career alerts and career-related events to students based on their specific interests. Emphasizing decentralized processing, "Bronco-Career Alerts" 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 connects students to relevant career opportunities, fostering a network that dynamically aligns with their professional ambitions and academic growth.**

I. MOTIVATION

University life is bustling with valuable events, but overwhelmed students often miss out due to mass email clutter. Our project aims to revolutionize communication, introducing a personalized publish-subscribe system. By tailoring event notifications to individual interests, we strive to break free from the one-size-fits-all approach, ensuring students receive information relevant to their academic and professional goals, fostering a more streamlined and enriching university experience.

II.DISTRIBUTED SYSTEMS CHALLENGES

The development and implementation of Bronco-Career Alerts introduce a distributed system designed to enhance communication within a university environment, particularly focusing on the orchestration of events, notifications, and user preferences. While the project aims to bridge the gap between students and valuable opportunities, it is essential to address and navigate various challenges inherent to distributed systems. Here, we outline key challenges and strategies to mitigate them.

A. Decentralized Communication

Universities are dynamic ecosystems with a multitude of events happening simultaneously. Coordinating and disseminating information across decentralized networks pose challenges in maintaining consistent communication.The publisher-subscriber model implemented in Bronco-Career Alerts leverages decentralized communication to allow students to subscribe to specific event types. This design helps in delivering personalized information efficiently.

B. Information Overload

Traditional communication methods, such as mass emails, often lead to information overload. Students may receive notifications about events unrelated to their interests, causing them to overlook critical opportunities.The publish-subscribe architecture, influenced by the literature review, allows students to tailor their event preferences. This personalized approach reduces information overload, ensuring students receive notifications relevant to their academic and career goals.

C. Scalability and Performance

As the number of students and events increases, ensuring scalability and maintaining system performance become critical challenges in distributed systems.The system architecture incorporates principles from efficient publish/subscribe models discussed in the literature review.

D. Fault Tolerance and Reliability

Distributed systems must be resilient to faults and failures to provide continuous service. Downtime or disruptions may result in missed opportunities for students.The use of Kubernetes and RabbitMQ ensures fault tolerance. If one instance fails, others can seamlessly take over, contributing to the system's reliability.

E. Consistency and Event Ordering

Ensuring consistency in the order of events and notifications is crucial for providing an accurate and reliable user experience.The publish-subscribe pattern, combined with proper concurrency control and mutual exclusion algorithms, maintains consistency and orderly event processing even in scenarios of high concurrency.

F. Integration with External APIs

Fetching and integrating data from external APIs, such as Greenhouse Job Board 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, Bronco-Career Alerts aims to provide a robust and user-friendly platform that effectively connects students with valuable opportunities while maintaining the reliability and scalability required in a university setting.

III. LITERATURE REVIEW

The Bronco Career Alerts project, which incorporates a publish/subscribe (pub/sub) messaging system, has benefitted from a wealth of research outlined in various IEEE papers. The insights from these papers have contributed significantly to the design and implementation of the project's pub/sub system.

A. Pub/Sub Communication Based on Subscription Aging

This concept inspired the implementation of subscription management, ensuring that the system remains efficient even as the number of subscribers grows. By integrating a similar model, the Bronco Career Alerts system mitigates potential bottlenecks, thereby maintaining high performance and reducing the load on the system.

B. An Efficient Publish/Subscribe System

The techniques from PUBSUB influenced the development of smart data structures and matching algorithms within the project, ensuring that the system performs optimally, especially when handling uniform subscription loads. This has been crucial for achieving scalability and efficiency in the Bronco Career Alerts system.

C. Survey of Publish/Subscribe Middleware Systems

This survey provided a clear understanding of the publish/subscribe architecture, highlighting its benefits over client-server models, which guided the architectural decisions in the Bronco Career Alerts project. It underscored the importance of event-based communication, which the project implements to ensure eventual consistency across distributed services.

D. Storm Pub-Sub

The high-throughput event matching system described in this paper influenced the project's approach to efficiently handling a large number of events and subscriptions. The use of parallel processing techniques within the consumer service ensures that the Bronco Career Alerts system can scale and maintain performance as the load increases.

E. The Hidden Pub/Sub of Spotify

By examining Spotify's implementation of the pub/sub pattern, the project gained insights into handling large-scale workloads and traffic patterns, which were instrumental in modeling the BroncoCarrerAlerts pub/sub workload for optimal performance.

F. Large-Scale Distributed Computing Oriented Publish/Subscribe System

The design considerations from this paper provided guidance on enhancing the system's support for distributed computing environments, which is evident in the project's use of Kubernetes for orchestration and RabbitMQ for message brokering.

By incorporating these research findings, the Bronco Career Alerts project has built a robust pub/sub system that is capable of handling the dynamic nature of real-time distributed applications. The system is scalable, efficient, and fault-tolerant, with an adaptable architecture that can accommodate an increasing number of users and complex workloads. The project's producer and consumer services utilize these principles to ensure that messages are published and consumed reliably,

demonstrating the practical application of academic research to solve real-world problems in system design and implementation.

IV. PROJECT DESIGN

In the architecture shown in Fig. 1, the RabbitMQ server, a crucial component of the system, is hosted on a Minikube cluster. Minikube is a tool that allows you to run Kubernetes locally, providing an efficient way to test and deploy applications in a containerized environment similar to a production setup. In this scenario, RabbitMQ benefits from the scalability and manageability offered by Kubernetes, enhancing its robustness and reliability as a message broker.

The producer, implemented in producer.py, is a Flask server running on localhost at port 5000. This server is responsible for publishing and broadcasting messages to various topics on the RabbitMQ server. On the other side, the consumer, encapsulated in consumer.py, operates on localhost at port 5001. It handles subscriptions, unsubscriptions, and the retrieval of messages from the RabbitMQ server.

Both the producer and consumer applications are configured to communicate with the RabbitMQ server hosted on the Minikube cluster. This setup ensures a seamless interaction between the Flask applications and the message broker, despite being in different environments.

To facilitate this communication, RabbitMQ is exposed using kubernetes service. This approach allows external applications, like our Flask servers running on localhost, to access the RabbitMQ server within the Minikube cluster. The service acts as a bridge, forwarding requests and responses between the Flask applications and RabbitMQ, ensuring a smooth flow of messages in 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.

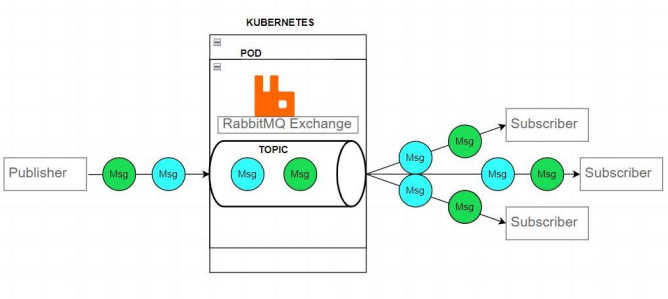

Fig. 1 High Level Architecture

V. EVALUATION

The Bronco Career Alerts system stands as a robust example of distributed messaging, featuring well-designed algorithms for broadcasting, replication, concurrency, and mutual exclusion. This advanced platform utilizes these algorithms to enhance fault tolerance, improve performance, and ensure scalability, while also securing and maintaining consistent communication throughout a decentralized pub/sub network.

A. Broadcast algorithm implementation

In a pub/sub (publish/subscribe) system, broadcasting refers to the ability to send a message to all subscribers, regardless of their individual topic subscriptions. This is particularly useful for disseminating urgent or critical information, such as alerts or notifications, which must reach a wide audience instantaneously. Our system originally supported topic-based messaging. To enhance this system with broadcasting capabilities, we made strategic modifications to the existing infrastructure, allowing it to handle both regular topic messages and broadcast messages.

We defined a special broadcast topic, labeled 'broadcast'. This unique identifier is used to route broadcast messages to all subscribers.We added a new endpoint, /broadcast, in the Flask producer application. This endpoint accepts broadcast messages.Upon receiving a broadcast message, the application publishes it to the RabbitMQ exchange with the 'broadcast' routing key.Each subscriber's queue is configured to bind not only to their specific topic but also to the broadcast topic. This is achieved by using the RabbitMQ queue_bind function.The consumer logic is adapted to handle and differentiate between broadcast messages and regular topic-specific messages. This ensures that

subscribers can process and respond to broadcast messages appropriately.

1) Process Flow: When an urgent message needs to be broadcasted, it is sent to the /broadcast endpoint of the Flask application in producer.py.The Flask application, acting as the producer, publishes this message to the RabbitMQ exchange with a routing key specific to broadcasting.RabbitMQ then delivers this message to all queues bound to the broadcast routing key. Since each subscriber's queue is bound to this key in addition to their specific topics, all subscribers receive the broadcast message. Each subscriber (or consumer) retrieves and processes the message. The consumer logic ensures that the broadcast message is recognized and handled as needed. By using a special broadcast topic within the same exchange, we avoid the complexity of managing multiple exchanges or altering the core architecture of the messaging system.Subscribers don't need to subscribe to additional topics or exchanges to receive broadcast messages.

B. Replication

In the Bronco Career Alerts project, a robust replication strategy is integral to the system's architecture, enhancing its fault tolerance and availability. This strategy is implemented using Kubernetes, which orchestrates containerized instances of the RabbitMQ message broker across a cluster. The system's resilience is fortified by deploying multiple replicas of RabbitMQ through Kubernetes Deployments, defined within a rabbitmq-deployment.yaml file, ensuring a predetermined number of Pod replicas are active at any given time. To manage state and message durability across these replicas, Kubernetes Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) are utilized, safeguarding against data loss during Pod restarts or rescheduling. Service discovery mechanisms provided by Kubernetes Services enable stable connectivity to the RabbitMQ broker, with traffic efficiently distributed among the replicas. Moreover, Kubernetes' health check probes monitor the health of RabbitMQ Pods, providing self-healing capabilities by automatically restarting non-responsive Pods, thus upholding the system's robustness. The combination of these features results in a messaging system that can withstand node failures and handle traffic surges without interruption, maintaining continuous processing capabilities. The implementation of this replication strategy within the Bronco Career Alerts system is a testament to Kubernetes' ability to provide a resilient and scalable infrastructure for distributed applications, ensuring high availability and reliability essential for the system's operational demands.

C. Concurrency and Mutual Exclusion

In the Bronco Career Alerts project, the interplay between threads and locks is central to managing concurrency and ensuring the system's robustness. Each consumer within the system is allocated its own thread, creating a multi-threaded environment where message processing can occur in parallel across multiple queues. This threading model significantly boosts the system's ability to handle high volumes of messages concurrently, leading to efficient workload distribution and scalability.

When a user subscribes to a topic, the system spins up a new thread dedicated to handling messages for that particular subscription. This ensures that the processing workload is spread across multiple threads, allowing for simultaneous, non-blocking operations. The independence of threads allows the system to be responsive and maintain high throughput even as the number of consumers grows.To complement the threading architecture, the system incorporates locks to manage mutual exclusion effectively. The threading lock, specifically the subscribe_lock, is a crucial component that the system uses to synchronize access to shared resources. This lock is particularly important during consumer subscription and unsubscription actions, which involve operations that can modify the state of the message queues. By locking around the code that binds or unbinds queues to the exchange, the system ensures that these critical sections are not executed concurrently, thus preventing race conditions and preserving data integrity. The use of locks in coordination with threads is a deliberate strategy that prevents any overlap in operations that could lead to inconsistencies or errors. When a thread needs to modify the state of the system, it must first acquire the appropriate lock, perform the operation, and then release the lock. This serialized access to sensitive operations guarantees that the system's state transitions are safe and consistent.In essence, the combination of threads for consumer message processing and locks for operation synchronization

reflects the system's sophisticated approach to concurrency. This dual strategy enables the Bronco Career Alerts project to maintain a highly reliable and efficient messaging service, ensuring that it can handle the dynamic and concurrent demands of its users effectively.

D. System Resilience and Efficiency

The Bronco Career Alerts project addresses several critical concerns in distributed systems:

1) Fault Tolerance: The system employs RabbitMQ's message queuing capabilities, which inherently provide fault tolerance through message persistence (delivery_mode=2) and durable exchange declarations. Kubernetes is used for container orchestration, which offers self-healing features such as automated restarts of failed containers, helping the system recover from faults seamlessly. Consumer threads are designed to handle connection errors and retry connections, ensuring continuous operation even in the event of temporary network issues.

2) Performance: Efficient message handling is ensured by RabbitMQ's advanced message routing and load distribution capabilities. Flask's lightweight framework enables quick and efficient handling of HTTP requests without significant overhead. The producer service efficiently interacts with external APIs and processes data to be sent to consumers without unnecessary delays.

3) Scalability: Kubernetes supports the system's scalability by allowing for horizontal scaling of services. The number of pod replicas can be adjusted based on the load, ensuring the system scales to meet demand. RabbitMQ's exchange and queue system can handle a growing number of messages and consumers without a drop in performance, vital for maintaining scalability.

4) Consistency: The system ensures eventual consistency, a common approach in distributed systems where all changes propagate through the system over time, ensuring that all consumers eventually receive the correct data. Using exchanges and binding keys in RabbitMQ ensures that messages are consistently routed to the correct queues.

5) Concurrency: Concurrency is managed using threading in the consumer service, allowing multiple consumers to operate in parallel without blocking each other. Locks (subscribe_lock) are employed to ensure that operations like subscribing and unsubscribing, which modify the state of the system, are thread-safe and free from race conditions.

6) Security: The project uses HTTP Basic Auth for interacting with RabbitMQ's management API, providing a simple authentication mechanism.Communication with external services, such as the Greenhouse jobs API, is performed over HTTPS, ensuring data in transit is encrypted. Bronco Career Alerts project incorporates a range of strategies to address common distributed system concerns, from fault tolerance and performance to scalability, consistency, concurrency, and security. These mechanisms work in tandem to create a resilient, efficient, and scalable system that maintains data integrity and provides a secure environment for users and data.

VI. IMPLEMENTATION DETAILS

The system is centered around two primary applications that are developed with Python Flask: producer.py and internal.py, which interact with a RabbitMQ server acting as the message broker. This setup is a classic example of a publish/subscribe (pub/sub) model, which is highly effective for handling event-driven communication in distributed systems.

A. Flask for Application Development

Flask is a lightweight and flexible web framework for Python, widely acclaimed for its simplicity and elegance. It is part of the microframework category, meaning it requires little to no boilerplate code for simple applications, making it an ideal choice for small to medium web services or APIs.

One of Flask's core strengths is its extensibility. While it provides only the fundamental tools needed to get a web application running (such as routing requests and handling responses), it can be easily extended with numerous extensions available for tasks like database integration, form validation, user authentication, and more. This allows developers to add only the components they need, keeping their application lightweight and efficient.

Flask's design is centered around simplicity and minimalism. Flask's built-in development server and debugger, support for unit testing, and RESTful request dispatching are features that made it a practical choice for our application.

In the context of producer.py and consumer.py, Flask's ability to quickly set up endpoints for HTTP requests is particularly useful. It enables the rapid development of web services for publishing and subscribing to messages in a pub/sub system, as demonstrated in the code. The Flask applications

interact with RabbitMQ, showcasing how Flask can be integrated into more complex architectures involving message brokers and distributed systems.

B. The Publisher

At the heart of producer.py lies its ability to publish messages to various topics. This Flask application establishes a connection to RabbitMQ, a powerful message broker, using pika, a RabbitMQ client library for Python. The connection parameters are set to ensure a reliable communication channel, including host, port, credentials, and heartbeat settings.

Once a connection to RabbitMQ is established, the application defines key elements for message routing within the RabbitMQ environment. These elements are the EXCHANGE_NAME, the BROADCAST_TOPIC, and the use of a routing key.

1) Exchange Name (EXCHANGE_NAME): In RabbitMQ, an exchange is a message routing agent, responsible for receiving messages from producers and pushing them to queues based on certain rules. In this application, the exchange named 'routing' is declared. The type of this exchange is 'topic', which means it routes messages to queues based on a pattern matching between the message's routing key and the pattern used to bind a queue to an exchange.

2) Broadcast Topic(BROADCAST_TOPIC): The application defines a special topic for broadcasting messages. The 'broadcast' topic is used to send messages to all subscribers regardless of their specific topic subscriptions. This is particularly useful for sending urgent or general notifications that need to be disseminated to all connected parties.

3) Routing Key: The routing key is a message attribute the exchange looks at when deciding how to route the message to queues (which are bound with a routing pattern). In the context of this Flask application, the routing key is dynamically set based on the 'topic' attribute of the incoming message. This allows for flexible and targeted message delivery, as messages can be routed to specific queues handling certain topics or to the broadcast topic for a wider reach.

Additionally, the application includes a function called fetch_external_data(). This function plays a crucial role in fetching data from an external source, specifically from an API endpoint provided by the Greenhouse job board for Discord. When called, the function makes an HTTP GET request to the specified URL. If the response is successful (HTTP status code 200), it parses and returns the JSON data from the API. This external data can then be used for publishing to the 'external' topic, thereby enabling the system to integrate and disseminate information from outside sources.

The application exposes two main endpoints: /publish and /broadcast. The /publish endpoint is versatile, catering to both internal and external topics. If the user is subscribed to an “internal”topic, it reads events from a file called internal_events.txt and publishes them to the designated “internal” topic. When dealing with an “external” topic, it fetches data from an Greenhouse Job Board API and then publishes it. The endpoint discerns between internal and external data based on the topic specified in the POST request.The payload for POST requests targeting either internal or external topics is structured in a JSON format. This payload comprises two key-value pairs: one for the message content and the other specifying the topic type. An example payload for a request is as follows:

```
{
  "topic": "internal",
  "events": [
     "Executives in house",
     "Networking:Panel discussions/events",
     "Technical Careers Meetup",
     "Careers in Business Meetup",
     "SCU Alumni Panel",
     "Finance in Tech Meetup",
     "Grab a COffee/Coffee chats"
  ]
}
```

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

In this JSON structure, "message" represents the data or information to be published, while "topic" indicates whether the message pertains to an internal or external event. This format ensures a

clear and structured way of transmitting data to the respective HTTP endpoint, facilitating efficient processing and routing of messages within the pub/sub system.

```
{
  "message": "Canceled: STEM Coffee Chat with Synopsys"
}
```

On the other hand, the /broadcast endpoint is designed for urgent communications that need to be disseminated widely, using a predefined broadcast topic.

C. The Subscriber - consumer.py

consumer.py complements producer.py by managing subscriptions and message retrieval. This Flask application also sets up a connection to RabbitMQ, ensuring subscribers can listen to the messages being published. It offers two primary endpoints: /subscribe, /unsubscribe.

The /subscribe endpoint allows users or services to subscribe to specific topics. Upon subscription, a unique queue, named after the user, is created and bound to the requested topic. This queue also binds to the broadcast topic, enabling the user to receive both specific and broadcasted messages. The /unsubscribe endpoint performs the opposite action, removing the binding between the user's queue and the specified topic. The payload for POST requests targeting either subscribe or unsubscribe endpoints is structured in a JSON format. This payload comprises two key-value pairs: one for the username and the other specifying the topic type. An example payload for a request is as follows:

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

D. The Role of RabbitMQ

Central to this architecture is RabbitMQ, which acts as the message broker. It handles the routing of messages from the publisher to the appropriate subscriber queues based on the topic. RabbitMQ's robustness allows for handling high volumes of messages and ensures reliable delivery, which is crucial in a distributed pub/sub system.

E. Sequence Diagram

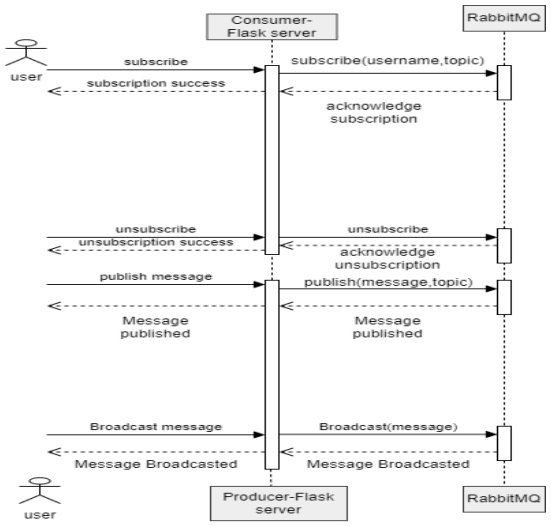

Fig. 2 Sequence Diagram

The sequence diagram in Fig. 2 depicts the interactions within the Bronco Career Alerts messaging system, emphasizing the communication between users, the Consumer-Flask server, the Producer-Flask server, and RabbitMQ. This diagram captures the flow of actions triggered by users and how these actions are handled by the system components.

From Fig 2, the sequence initiates with a user sending a subscription request to the Consumer-Flask server, which in turn interacts with RabbitMQ to register the subscription. Once successful, an acknowledgment of the subscription is sent back to the user. Similarly, the process for unsubscribing follows a user's request to the Consumer-Flask server, with an acknowledgment of unsubscription success returned upon completion. Furthermore, the diagram illustrates the process of publishing a message, where the user interacts with the Producer-Flask server that communicates with RabbitMQ to publish the message to a specific topic. An acknowledgment of the published message is then conveyed back to the user. Lastly, the sequence delineates the broadcasting of a message, where the Producer-Flask server sends the broadcast to RabbitMQ, which then disseminates the message to all subscribers, showcasing the

system's capability to handle broad and targeted communication efficiently.

F. Class Diagram

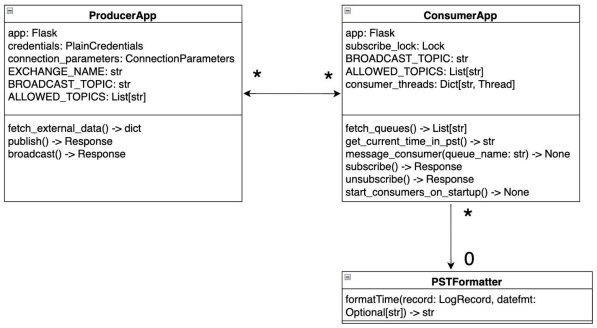

Fig. 3 Class Diagram

The class diagram presented in Fig. 3 serves as an object-oriented model for the Bronco Career Alerts project, delineating the structural framework of the system's backend components. It explains the relationships and interfaces between the ProducerApp and ConsumerApp, which collectively manage the core functionality of the messaging service. The ProducerApp class is characterized by its Flask application instance, RabbitMQ credentials, connection parameters, and methods for fetching external data, publishing messages, and broadcasting to all subscribers. Notably, it maintains a list of allowed topics, ensuring message publishing adheres to the system's predefined topics.

The ConsumerApp class, also a Flask application, encompasses a subscription lock for thread safety and manages a dictionary of consumer threads, allowing for parallel message consumption. It includes methods for queue retrieval, subscription and unsubscription handling, and the initiation of consumer threads upon system startup.

Additionally, the PSTFormatter class provides a specialized method for time formatting within log records, facilitating consistent time-stamping across the system's logging processes.

This diagram is crucial for understanding the project's backend architecture, reflecting a system designed for efficient pub/sub communication with a focus on modularity and process encapsulation.

VII. DEMONSTRATION OF SYSTEM RUN

The producer service successfully fetches external job postings from the Greenhouse API and publishes external events to the designated user as shown in Fig. 8 and similarly internal events are published through API (Fig. 9) are published (Fig. 10). Both internal and external events with persistence guaranteed (Fig. 14) ensuring message durability.

The broadcast functionality is confirmed to work as intended, delivering messages to all active consumers as shown in Fig. 12. The consumer service effectively manages user subscriptions, with the ability to start consumer threads dynamically on startup. Proper synchronization is maintained during subscribe and unsubscribe operations, ensuring that no race conditions occur even under concurrent requests.

Concluding, the demonstration of the Bronco Career Alerts system in a simulated Minikube environment (Fig. 13) showcases a fully operational messaging system, capable of handling real-world scenarios for career alert dissemination. The system's architecture, with its robust services for publishing and consuming messages, confirms its readiness for deployment. The successful run indicates that the system can reliably manage and route messages, providing a high quality service to its users and validating the effectiveness of using Kubernetes for orchestrating such distributed systems.

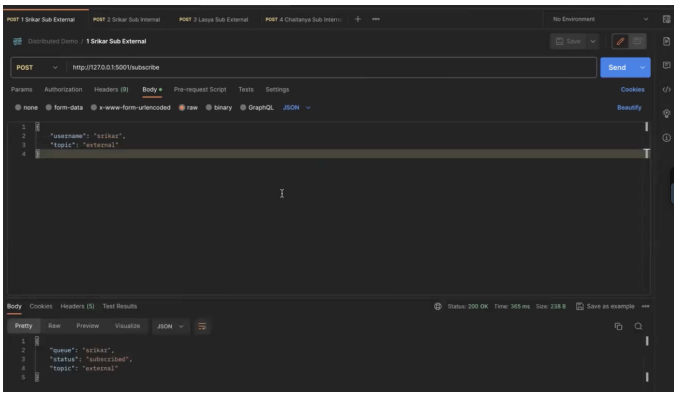

Fig. 4 Subscribing to external events

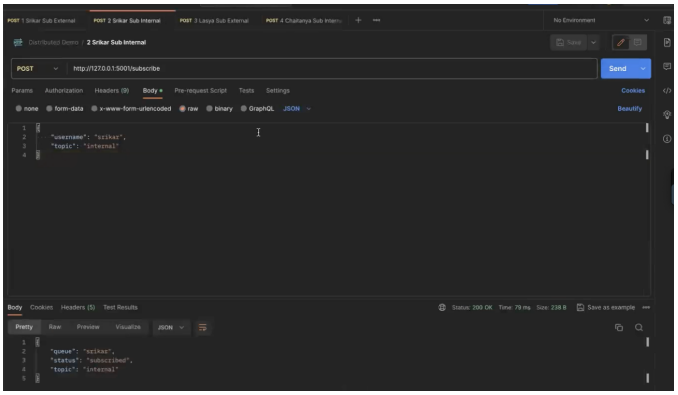

Fig. 5 Subscribing to internal events

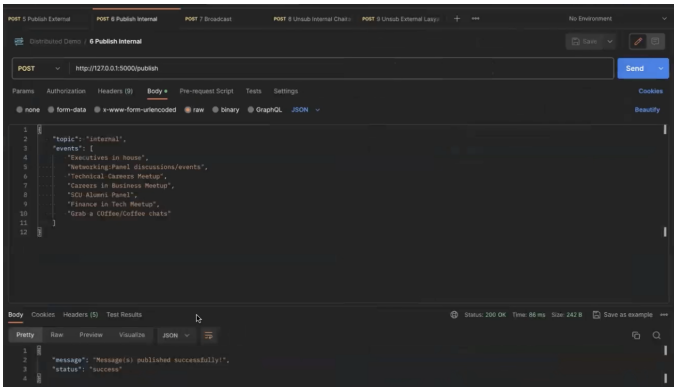

Fig. 9 Publishing Internal Events

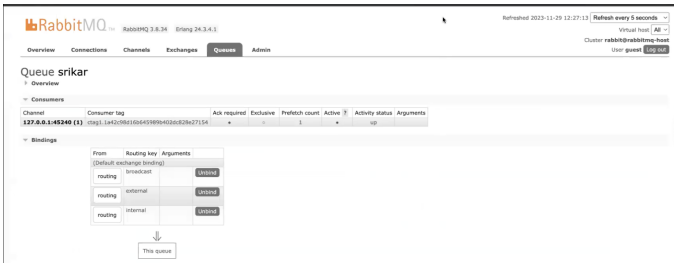

Fig. 6 Queues Created in RabbitMQ UI

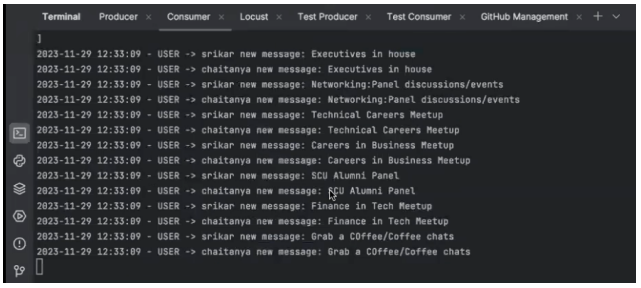

Fig. 10 Displaying Internal Events

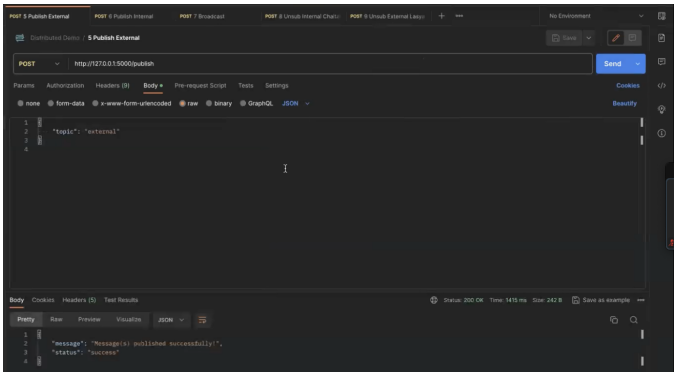

Fig. 7 Publishing External Events

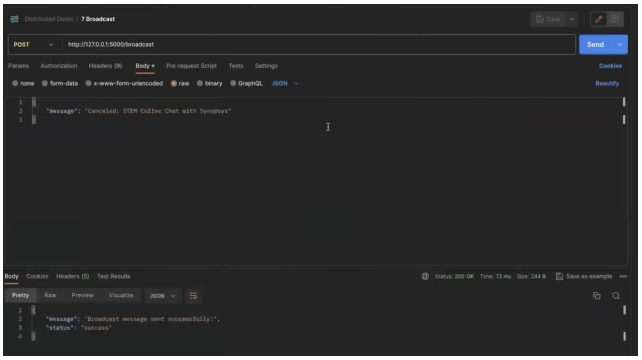

Fig. 11 Broadcasting events

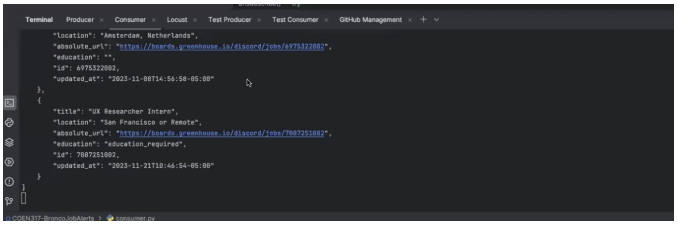

Fig. 8 Displaying External Events

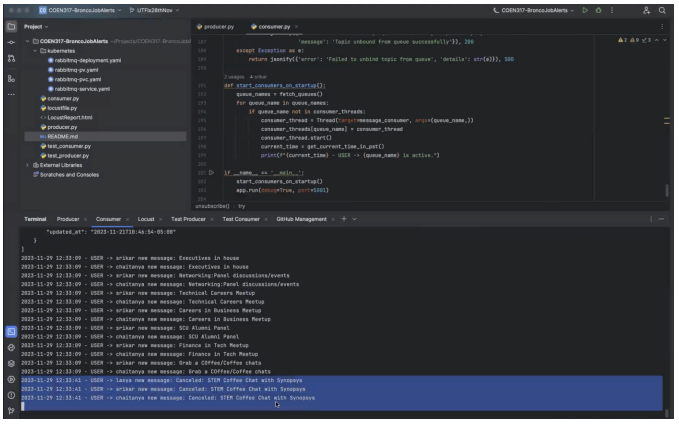

Fig. 12 Displaying broadcasted events

```
saisrikar@sais-MacBook-Pro COEN317-BroncoJobAlerts % kubectl delete pod rabbitmq-ccd658c56-f2zcn
pod "rabbitmq-ccd658c56-f2zcn" deleted
saisrikar@sais-MacBook-Pro COEN317-BroncoJobAlerts % kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
rabbitmq-ccd658c56-4qb2m   2/2     Running   0          9s
```

Fig. 13 Deleting Pod

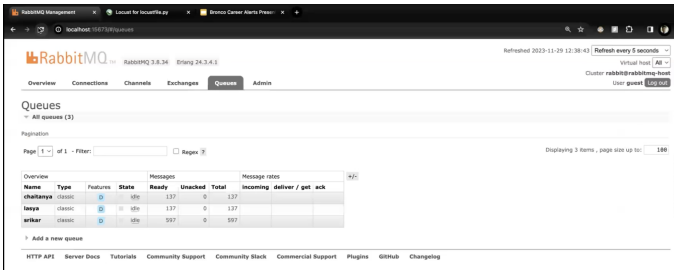

Fig. 14 Messages persisted after POD Restart

VIII. PERFORMANCE ANALYSIS

Locust is an open-source load testing tool written in Python. It is used for testing the performance of web applications and APIs under various load conditions. Locust is unique in its design because it allows writing test scenarios in Python, offering great flexibility and the ability to script complex user behaviors. It simulates users accessing your system, enabling you to see how it handles increased traffic and to identify bottlenecks.It can simulate millions of simultaneous users, providing insights into how a system behaves under high concurrency.Provides a web interface to monitor the test's progress in real-time.

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.

Each user class has its host and wait time set. The host is the base URL of the application being tested, and the wait time is the pause between each task execution.When the script runs, Locust creates instances of ProducerUser and ConsumerUser. These instances perform their tasks, hitting the defined endpoints with HTTP requests.The real-time monitoring interface of Locust allows observing the number of requests per second, response times, and the number of failures, providing a clear picture of how the system performs under load. By using Locust, we can effectively simulate various user interactions with your pub/sub system, giving us valuable insights into its performance characteristics. 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.

Request Statistics

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/broadcast	1632	0	984	24	4107	79	9.3	0.0
POST	/publish	3213	1	1173	29	4250	92	18.3	0.0
POST	/subscribe	1181	0	3693	29	10244	72	6.7	0.0
POST	/unsubscribe	1281	0	3506	27	12826	128	7.3	0.0
	Aggregated	**7307**	**1**	**1947**	**24**	**12826**	**92**	**41.5**	**0.0**

Fig. 15 Request Statistics

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/broadcast	800	1000	1200	1500	2100	2600	3300	4100
POST	/publish	1000	1200	1400	1700	2300	2900	3700	4300
POST	/subscribe	3900	4200	4700	5300	6700	7300	8800	10000
POST	/unsubscribe	3700	4000	4500	5400	6500	7000	8200	13000
	Aggregated	**1300**	**1600**	**2500**	**3500**	**4600**	**6000**	**7700**	**13000**

Fig. 16 Response time statistics

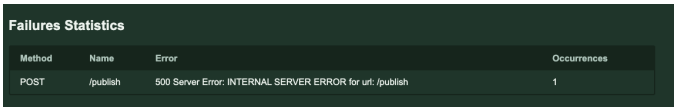

Fig. 17 Failure statistics

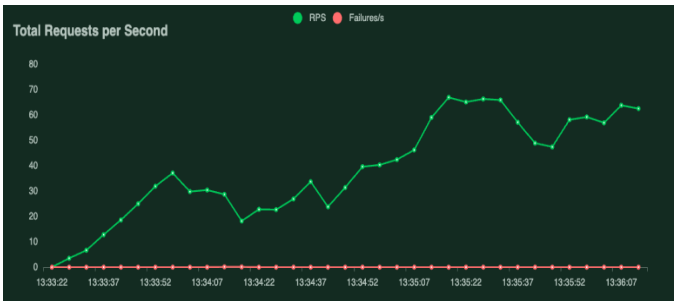

Fig. 18 Total requests per second

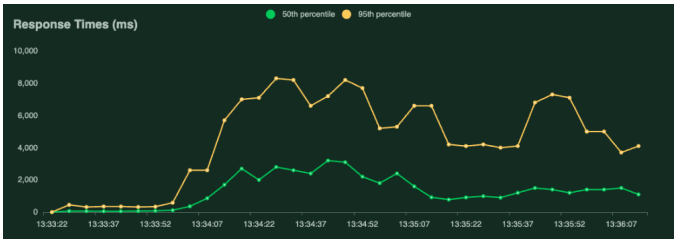

Fig. 19 Response times

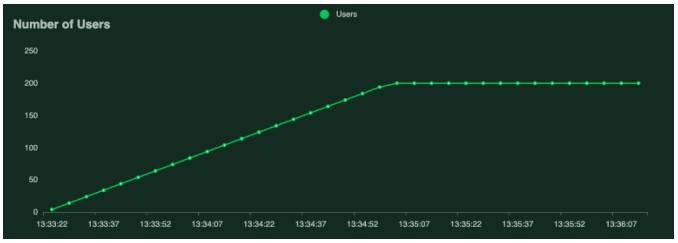

Fig. 20 Number of Users

The Locust performance test report indicates the system tested a mix of API endpoints with HTTP POST methods in Fig. 15. The /publish endpoint had one failed request (Fig. 17), while the others had none. The average response time across all endpoints was 1,947 milliseconds (Fig. 16).

Response times varied (Fig. 19), 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. 20) 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 summary, the test demonstrated robust performance for the majority of the test, with a single endpoint /publish showing a failure, suggesting a specific area that may need attention for improvement. The percentile breakdown of response times provides insights into the system's behavior under load, with higher response times at the 95th and 99th percentiles, highlighting the system's performance boundaries.

IX. TESTING RESULTS

The Bronco Carrer Alerts system has undergone a series of unit tests to validate the functionality of its producer and consumer services. These tests are designed to ensure that each unit of the code performs as expected in isolation. The following report summarizes the unit testing outcomes and outlines the 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 Greenhouse jobs 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.21 all tests are passed, indicating that the producer service is functioning correctly.

```
saisrikar@sais-MacBook-Pro COEN317-BroncoJobAlerts % python3 -m unittest -v test_producer.py

test_broadcast_success (test_producer.TestFlaskApp.test_broadcast_success) ... ok
test_fetch_external_data_success (test_producer.TestFlaskApp.test_fetch_external_data_success) ... ok
test_publish_internal_topic_success (test_producer.TestFlaskApp.test_publish_internal_topic_success) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.019s

OK
```

Fig. 21 Unit test results for producer

2) Consumer Service Tests:

Test Start Consumers on Startup: This test confirmed that consumers (queues) are active upon the system's startup.
Test Subscribe Success: The subscription service was tested to ensure users could subscribe to topics successfully.
Test Unsubscribe Success: The functionality to unsubscribe from topics was validated to ensure users could cease receiving messages from specific topics.
Outcome: As shown in Fig.22 all tests are passed, demonstrating that the consumer service is operating as intended.

Fig. 22 Unit test results for 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): User testing will simulate publishing events fetched from the external Greenhouse jobs API to ensure the system accurately retrieves and disseminates job listings.
Test cases will include various response scenarios from the Greenhouse API to validate the system's resilience to external data changes.
2) Producer Internal (POST /publish): The internal publishing endpoint will be tested with multiple events to ensure the system correctly handles and routes a series of internal messages.
Simulations will involve checks for message persistence and order of delivery.
3) Broadcast (POST /broadcast): Tests will ensure that broadcast messages reach all active users, with validations on both the producer and consumer sides.
Stress testing may be performed to evaluate the system's performance under high load.
4) Consumer Subscribe (POST /subscribe): User tests will involve subscribing to both internal and external topics, with verifications for successful queue bindings and message receipts.
Edge cases, such as subscribing to non-existent topics or invalid usernames, will also be evaluated.
5) Consumer Unsubscribe (POST /unsubscribe): This endpoint will be tested to confirm that unsubscribing effectively prevents message delivery to the user's queue.
Additional tests will ensure that the unsubscription process does not inadvertently affect other users or queues.

The unit tests have confirmed that the individual components of the Bronco Career Alerts system function correctly. The subsequent user testing of the system's API endpoints will provide further assurance that the system behaves as expected in real-world scenarios, offering a robust platform for job alerts dissemination. The testing framework, with its comprehensive coverage, ensures that the Bronco Career Alerts system can reliably manage and route messages, maintaining high service quality for its users.

X. CONCLUSION AND REFLECTIONS

The Bronco Career Alerts project stands as a testament to the practical application and inherent benefits of the publish/subscribe paradigm in distributed systems. The system's development journey was a deep dive into the complexities of distributed architectures, where the merits of topic discovery and scalability had to be designed rather than inherited through rigorous research and implementation of various protocols and algorithms

Critical to the system's robustness is the use of RabbitMQ for message queuing, ensuring fault-tolerant communication, and Kubernetes for orchestration, providing self-healing and scalability. The system's design employs the broadcast algorithm for efficient message dissemination and the replication strategy to enhance data integrity and availability. Concurrency and mutual exclusion are adeptly managed through threading and locking mechanisms within the consumer service, safeguarding against race conditions and ensuring thread-safe operations.

In essence, Bronco Career Alerts is more than just a career alerting service; it's a robust, scalable solution that encapsulates the core principles of distributed systems, demonstrating exceptional fault tolerance, performance, and concurrency management—making it an exemplary model in the educational and professional domains.

XI. REFERENCES

[1] *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, pp. 475-479, doi: 10.1109/IISR.2018.8535938.*

[2] *T. B. Mishra and S. Sahni, "PUBSUB: An efficient publish/subscribe system," 2013 IEEE Symposium on Computers and Communications (ISCC), Split, Croatia, 2013, pp. 000606-000611, doi: 10.1109/ISCC.2013.6755014.*

[3] *S. Kul and A. Sayar, "A Survey of Publish/Subscribe Middleware Systems for Microservice Communication," 2021 5th International*

Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), Ankara, Turkey, 2021, pp. 781-785, doi: 10.1109/ISMSIT52890.2021.9604746.

[4] *M. A. Shah and D. B. Kulkarni, "Storm Pub-Sub: High Performance, Scalable Content Based Event Matching System Using Storm," 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, Hyderabad, India, 2015, pp. 585-590, doi: 10.1109/IPDPSW.2015.95.*

[5] *Setty, Vinay & Kreitz, Gunnar & Vitenberg, Roman & van Steen, Maarten & Urdaneta, Guido & Gimaker, Staffan. (2013). The hidden pub/sub of spotify. 231-240. 10.1145/2488222.2488273.*

[6] *J. G. Ma, T. Huang, and J. L. Wang, "Underlying Techniques for Large-Scale Distributed Computing Oriented Publish/Subscribe System", Journal of Software, vol. 17, pp. 134-147, Jan. 2006.*

XII. DIVISION OF WORK

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