SANTA CLARA UNIVERSITY

School of Engineering

Distributed Emergency and Traffic incident Notification System

Naveena Avula (07700009231)

Sri Sai Saketh Chillapalli (007700000089)

Karumanchi Samuel (07700009611)

Introduction

Background: Emergency Situations

- **Public Safety:** In today's interconnected world, the rapid and accurate dissemination of emergency information and traffic incidents is crucial for public safety.

Challenge: Notification Overload

- **Notifications Everywhere:** 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.

Solution: Distributed Emergency and Traffic Incident Notification System

- **Smart Fix:** The proposed Distributed Emergency and Traffic Incident Notification System aims to address these challenges by leveraging cutting-edge distributed systems technology and a publisher-subscriber model.

- **User Power:** Users get to choose what they want to hear about, so they only get info that matters to them.

Related Work & Challenges

Adaptable Models of Communication:
- The necessity of dynamic communication in large systems is emphasized by research.
- Subscription aging is used by the "Distributed Emergency and Traffic Incident Notification System" to improve performance.

Survey of Microservice Communication:
- Focuses on microservices pub/sub for real-time applications.
- Impacts the event-based communication strategy of the "Distributed Emergency and Traffic Incident Notification System".

Challenges:
- Overcoming Information Overload: Multiple notifications might lead to an excessive amount of information.
- Consistency in Event Ordering: Maintaining notifications in sequence.

Design Choices

- **Microservices Architecture:**

Utilizing Flask to create lightweight RESTful services (producer and consumer) aligns with the microservices approach, ensuring scalability and maintainability.

- **Messaging with RabbitMQ:**

Choosing RabbitMQ as the message broker allows for reliable and scalable asynchronous message processing.The direct exchange type was selected for specific routing between producers and consumers, enabling precise message delivery.

- **Containerization with Kubernetes:**

Deploying on Kubernetes suggests a decision for high availability, load balancing, automated deployment, and scaling.

- **Persistent volume configurations (kubernetes volumes):**

Ensure that data is retained across pod restarts.

- **Topic-Based Routing:**

Messages are routed based on topics, enabling selective message consumption.

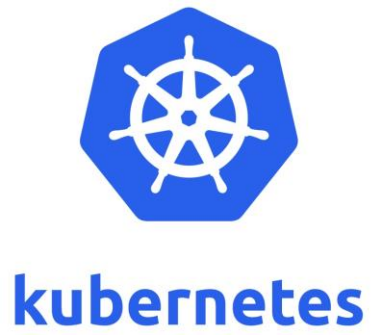
Challenges

- **Service Orchestration:** Several services can be difficult to coordinate. facilitating dependable and effective communication between services, particularly as the number of services increases.
- **Message Durability and Delivery Guarantees:** RabbitMQ can be configured to prevent message loss during transmission, especially in the case of service interruptions.
- **High Volumes:** RabbitMQ can be configured to process large message volumes without experiencing major delays or backlogs, and the load can be efficiently distributed across consumers to avoid overloading any one of them.
- **Debugging:** Although it can be difficult, tracking and monitoring the message flow via Flask and RabbitMQ services in a distributed system is crucial for troubleshooting.

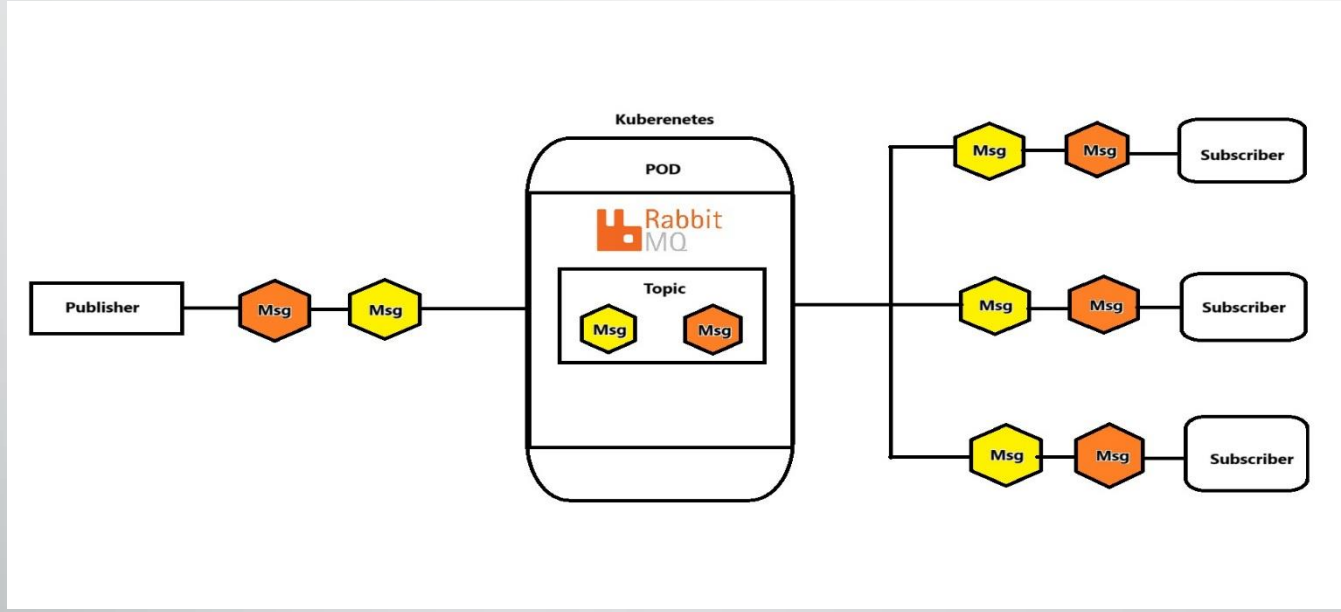
Algorithms Covered

- Mutual Exclusion
- Concurrency Control
- Broadcasting
- Replication & Consistency protocol

Techstack



Architecture-Overview



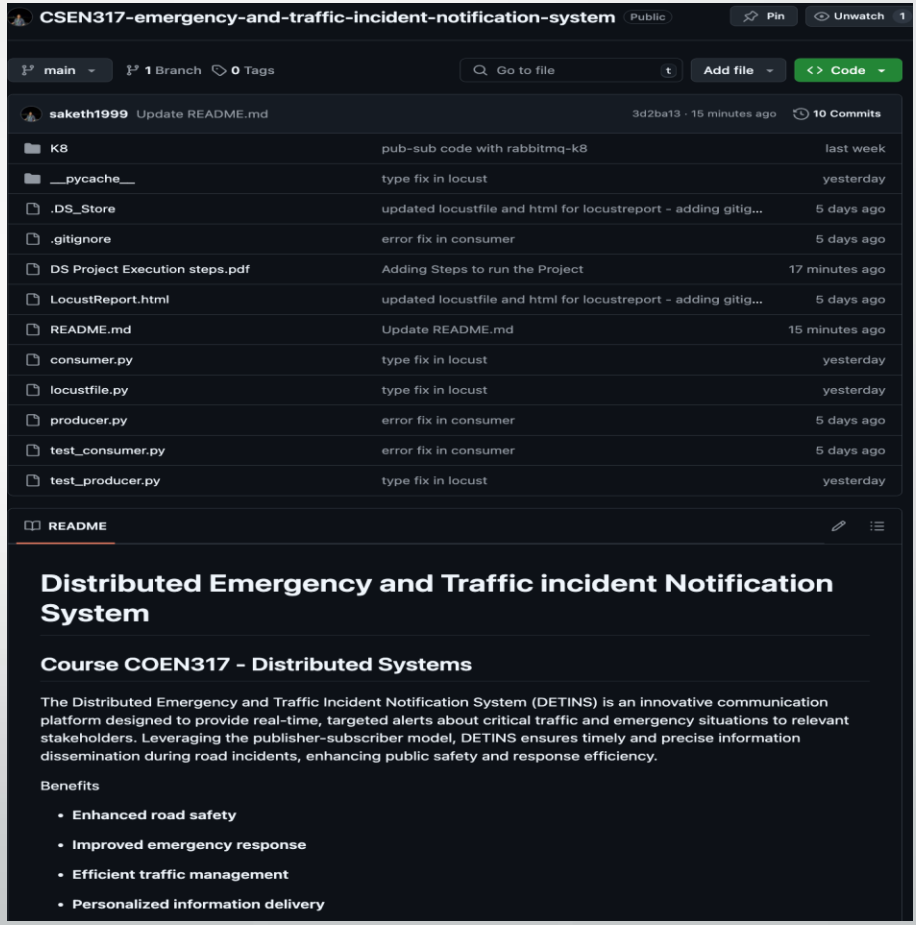
Kubernetes Objects

```
base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system/K8 git:(main) * minikube (0.148s)
kubectl get pv

NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                   STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
rabbitmq-pv   1Gi        RWO            Retain           Bound    default/rabbitmq-pvc    manual         <unset>                          7d14h

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system/K8 git:(main) * minikube (0.134s)
kubectl get pvc

NAME           STATUS   VOLUME        CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
rabbitmq-pvc   Bound    rabbitmq-pv   1Gi        RWO            manual         <unset>                 7d14h

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system/K8 git:(main) * minikube (0.114s)
kubectl get deployments

NAME       READY   UP-TO-DATE   AVAILABLE   AGE
rabbitmq   1/1     1            1           7d14h

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system/K8 git:(main) * minikube (0.124s)
kubectl get svc
NAME               TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)                         AGE
kubernetes         ClusterIP   10.96.0.1      <none>        443/TCP                         8d
rabbitmq-service   NodePort    10.106.17.29   <none>        5672:30672/TCP,15672:31672/TCP   7d14h

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system/K8 git:(main) * minikube (0.114s)
kubectl get pods
NAME                        READY   STATUS    RESTARTS      AGE
rabbitmq-764767fd85-2rxbs   1/1     Running   1 (47h ago)   47h

base ~/Desktop/myDSProject/CSEN317-emergency-and-traffic-incident-notification-system/K8 git:(main) * minikube
```

Repository Link

GITHUB:
https://github.com/saketh1999/CSEN317-emergency-and-traffic-incident-notification-system

Please follow the steps in - DS Project Execution pdf to run the project




LIVE
DEMO

Evaluation

- **Decoupled Microservices Architecture :** The design allows services to communicate asynchronously, reducing their dependence on each other. This will improve the system's maintainability and scalability.
- **Scalability:** RabbitMQ's capacity to handle high-volume and high-throughput messages is essential for real-time processing systems. Different queues and topics are used to assist scaling.
- **Load Testing with Locust:** The system is mostly stable, even though the one failure of the publish request suggests room for improvement.
- **Reliability and Performance**: To assist consumers share the burden equally, customer service uses Prefetch_count=1.

Metrics for Evaluation

- **Throughput:** The system appears to be able to manage a considerable load as measured in Requests Per Second (RPS), with a maximum RPS of 40 for aggregated requests.
- **Latency:** The system has variable latencies for different operations, according to the response time statistics. The subscribe and unsubscribe functions, for example, have longer maximum response times, which may suggest that they require more resources.
- **Error Rate:** The load test shows a low error rate, with only one publish operation failing, suggesting that the system is reasonably reliable under the given conditions.

Unit Testing

```
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
```

Load Testing using Locust

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

# 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')

Future Plans

Optimization and Tuning:

- The system may require optimization in light of the latency and the one failure that was noticed. This could entail fine-tuning Kubernetes resource allocation, optimizing Flask application code, or adjusting RabbitMQ setups.

Scalability Testing:

- The system's scalability will be tested by gradually raising the load until it reaches the system's maximum capacity, in addition to the present load tests. This will assist in determining when the system must scale out.

High Availability and Fault Tolerance:

- Priorities could include putting high-availability setups for RabbitMQ into place and making sure the Kubernetes deployment can withstand node failures without causing service interruptions.

Monitoring and Alerting:

- Establishing thorough monitoring and alerting for RabbitMQ and the Flask application within Kubernetes to obtain up-to-date information on the system's health and performance.

Summary

- Microservices Architecture
- Message Brokering with RabbitMQ
- Consumer Service
- Producer Service
- Load Testing with Locust
- Logging and Monitoring
- Error Handling and Reliability
- Deployment in Kubernetes
- Python and Flask