# Project Kafkaesque

## A publish-subscribe based distributed communication system


**Aarohi Agarwal**

**Computer Science**

**The George Washington University**

**Washington, DC, 20052**

**aagarwal14@gwu.edu**

**Biyas Basak**

**Computer Science**

**The George Washington University**

**Washington, DC, 20052**

**biyas@gwu.edu**

**Chinmay Patil**

**Computer Science**

**The George Washington University**

**Washington, DC, 20052**

**patchinmay@gwu.edu**

**Dilip Varma Sagi**

**Computer Science**

**The George Washington University**

**Washington, DC, 20052**

**dilipvarma7@gwu.edu**


## Introduction

We are in the 21st century and almost everything is available on our fingertips. Numerous amounts of software and different applications are running on our hand and generating a lot of data. With the usefulness and ease of availability of these applications and software, there is a great architecture responsible for smooth functioning. These modern days applications or software are usually built on the principles of distributed systems which makes them to function smoothly. One of the important features of any system is to data flow and communication between one component to another. There are various ways in which data can be moved in distributed systems such as using ETL, messaging systems, etc. Along with the communication and data flow the system also needs to perform flawlessly. Fault tolerance is one of the major concerns while building any distributed system (Aung, T., Min, H. Y., & Maw, A. H, 2019). Along with the fault-tolerance a good distributed system should also be easily scalable and can be easily decoupled.

This project will emulate a publish subscribe communication model for distributed systems such as Apache Kafka and RabbitMQ for collecting and distributing real-time data generated from multiple sources. As a part of implementation, a publisher-subscriber model for distributing messages will be created in which multiple consumers (VM's, servers, etc.) subscribe to a message queue which consists of data generated by the producer machines. This project also explores as to how to partition the messages across different segments and data replication to achieve fault tolerance.

## Project Design

The project aims to build a publish-subscribe type message broker system which internally uses Service-Oriented Architecture (SOA) consisting of multiple services, jobs and scripts in order to exchange messages between two computing systems. A typical implementation includes one-or-more publishers who put the message in a data structure known as message queue or broker and one-or-more subscribers that act as a client and consume the messages from the broker, processing them and sending the end-results back to the client that sent the request. All the individual components and interactions that take place between them are described as follows: -

1) **Publishers** : - The publishers can be any device that acts as a server that sends messages to the message queue/broker. These messages are filtered by topics which act as a named destination as to which broker the message will end up into. While sending messages, publishers do not have any knowledge of the subscribers or the clients that will consume these messages. By doing this, the publish-subscribe architecture helps to decouple different components of the system.

2) **Subscribers: -** The subscribers act as the receivers of the messages published by the publishers to the brokers. The subscribers receive the messages by subscribing to named destinations , where publishers send the messages in the broker. These named destinations can be either a message queue or a topic, which decides the order in which subscribers will receive the messages. The named destinations are in turn divided into partitions, and often subscribers form consumer groups related to a particular partition and subscribe only to topics present in that partition. This prevents the failure of a particular node in a partition, from affecting other clients in receiving messages, thereby guaranteeing at least once delivery of messgaes.

3) **Messages: -** A message forms a collection of bytes, to be transferred from the server to the client, while maintaining the integrity and consistency of data. The publishers and subscribers communicate with each other by passing messages, related to a topic, via a message queue or broker. Once the messages are published to the appropriate broker, they can be filtered by topic, content, type, etc. based on different subscription models.

4) **Message Brokers: -** The message brokers resolve the transparency challenge of the distributed message queueing system, by adding a layer of abstraction to the developers who wish to exchange messages between different services in a service oriented architecture. The message broker acts as an intermediate buffer between two services, which send and receive messages to and from each other using this data structure. This helps in decoupling of sending and receiving messages, as each service only needs to know connection details of the message broker and name of queue/topic to send and receive messages. This also helps to prevent Denial of Service (DoS) attacks, or the loss of messages in case of application crash, as messages are not removed from the buffer until the receiving application is ready to consume messages.

5) **Queues:-** The message queues are a type of First-In-First-Out data structures, which handle the ordering of messages by ensuring that they are received in the same order in which they are placed in the queue. The publishers and subscribers communicate by sending and receiving messages to and from message queues. Multiple producers can produce the messages and place them in a single message queue.Multiple subscribers can subscribe to a single message queue, and continuously poll the message queues, and once a subscriber has processed a message, it is removed from the message queue. The subscribers receive messages from the queue in a round-robin manner- that is from the list of subscribers who are ready to receive a message, the one that has least recently received a message will have a higher priority.

   **Applications of Queues in Distributed message queuing system: -**

   In a distributed message queueing system like RabbitMQ or Apache Kafka, the message queues may be used for the job scheduling, in which tasks are distributed to different servers capable of performing them. For instance, consider an image formatting cloud based application, in which images are posted to message queues, which is

subscribed by multiple servers. These servers continuously poll the message for new images, thereby enabling a pull-based distributed message queue system. When one worker processes an image, it saves the image to a local file server- before receiving the next image (Ref: Fig-1.0(b)). The image is removed from the queue once it is processed by a worker, to ensure that each image is processed exactly once. This helps to solve following challenges of distributed system:-

1) **Scalability: -** The rate of processing images increases proportionally to the increase in number of servers subscribed to a queue.
2) **Fault-tolerance: -** If any worker fails while processing an image, the message queue may redeliver the same image to another subscriber, after a visibility timeout, to ensure at least once processing of images.

6) **Topics: -** The topics are also FIFO data structures like queues, however, they ensure that messages are delivered at least once, instead of exactly once delivered as in case of queues. This is achieved by delivering messages to every subscriber that has subscribed to a particular topic. Thus, each subscriber receives a copy of each message that is sent to the queue, and any subscriber can process any message from the queue.

   **Applications of topics in Distributed message queuing system: -**

   The topics can be used for updating the information in a distributed cache. This can be used to resolve **consistency** and **replication** challenges of the distributed message queuing system. Let us consider a distributed cache system that maintains stock prices, and is updated in real-time using streaming data generated by mobile phones in different regions of the world. As queues only send messages to a single subscriber, in this case the stock price will not be updated to all the subscribers. Hence, topics can be used such that all the local copies of the stock database contain the same price across all the replicas. A topic would publish the price to each subscriber, thereby ensuring that all replicas have consistent data.

**Interaction between the components: -**

The server acts as a producer and places the request to process the message in a broker which can be either a message queue or a topic (Ref: Fig-1.0(b)). A topic is a stream of messages categorized by a particular type. This kind of model is called a push-based model where the streams of a message are pushed on one of the available brokers. For this implementation, at least one delivery of messages is considered, for ensuring high availability of the system, hence a copy of message will be delivered to all the subscribers of a topic.

The topics are then divided into partitions. The subscribers form consumer groups for related topics, and subscribe to a particular partition that contains that topic(Ref: Fig-1.0(a)). The subscribers, also known as workers then poll the message queue continuously for new messages. This kind of model is called a pull-based model where the message is retrieved from the broker When an message is placed in a queue, one of the subscribers will process the message and will remove it from the queue. If the response is not generated within a specified time, known as visibility timeout period, the image will be published by the producer again to that topic, and some other worker will process the message, thereby guaranteeing successful processing of the message in case of failures of worker nodes.

These partitions are in turn replicated to multiple brokers (message queues) to ensure fault tolerance. The replication factor indicates how many copies of partition are maintained for each topic. The message ordering is handled within replicated queues using leader-follower pattern. Each partition has a single leader to handle read and write requests and a configurable number of followers to replicate writes to leaders in the background. To avoid single point of failure, we will be implementing a resource manager instead of a single master node to initiate a new leader election in case of broker hosting the leader fails. This information is then propagated to all the clients subscribed to a topic in that partition.

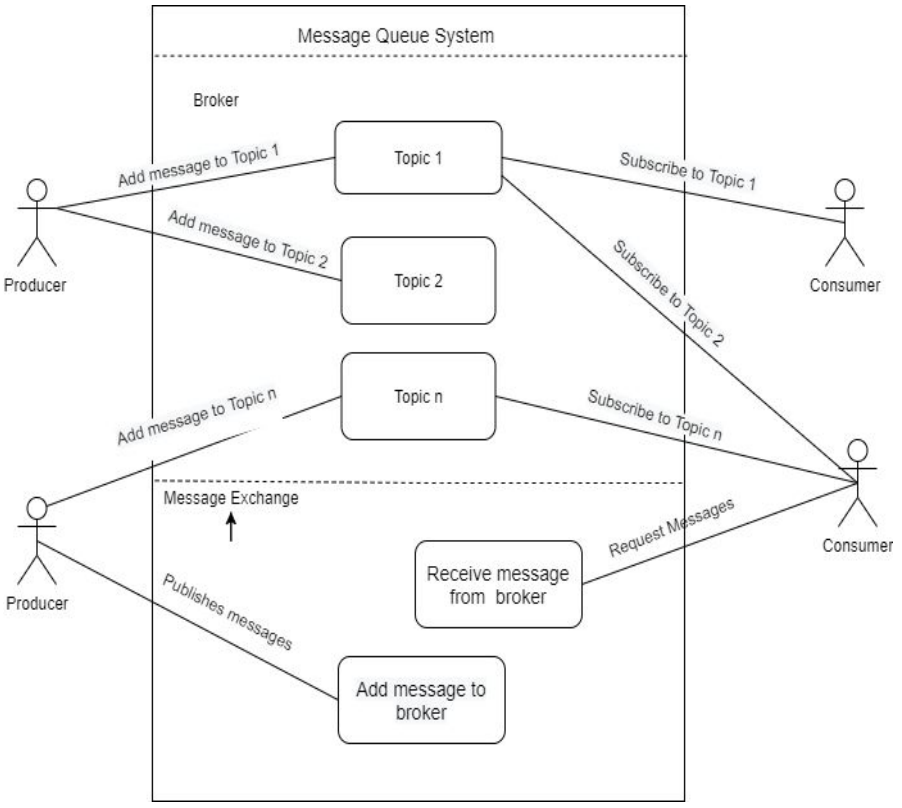



**Fig-1.0 (a)- Use case diagram for the distributed message queuing system**

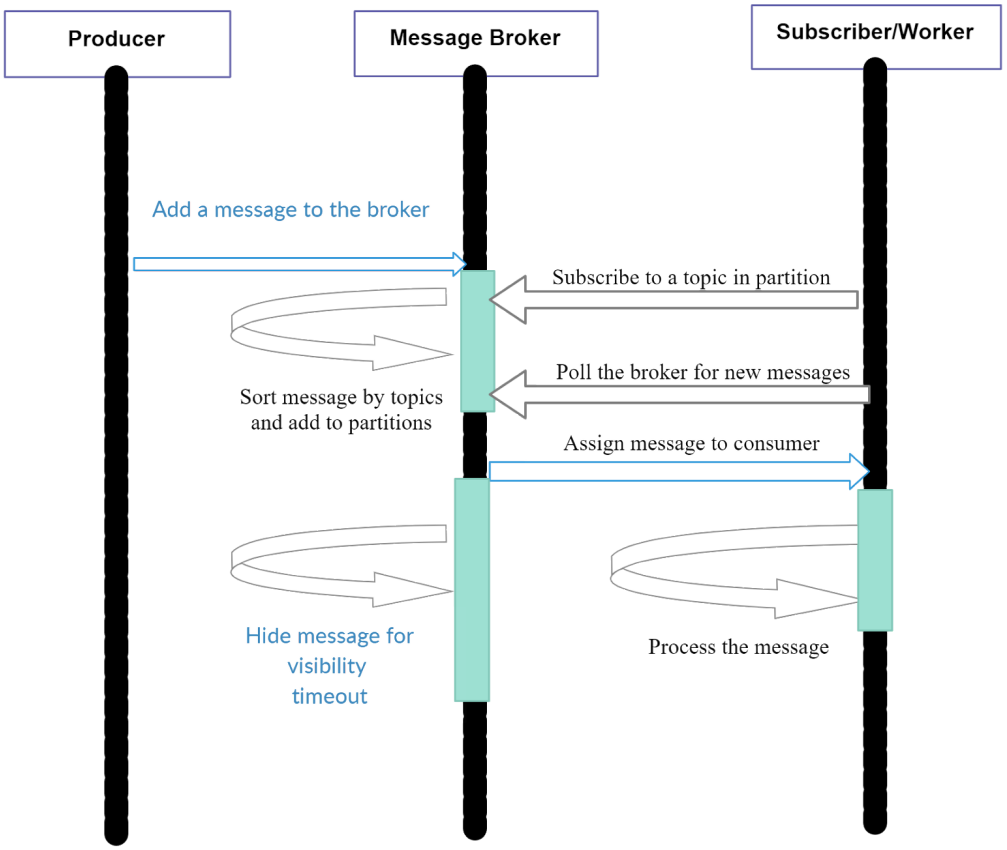



**Fig-1.0 (b)- Sequence diagram depicting steps performed by message queuing system for processing messages**

**Challenges**

To design and implement any distributed system it should be fault-tolerant, easily scalable, transparent, should also be decoupled easily, concurrent, etc. In this project we will be mainly focusing on challenges like how the system can be fault tolerant, how we can scale easily, good throughput, and how we can implement the decoupling of services in this System.

- **Decoupling of services**
  Decoupling the publishers and subscribers is arguably the most fundamental functionality of a pub/sub system (Dobbelaere, P., & Sheykh Esmaili, K. 2019).The pub-sub messaging framework provides a broker that both producer and consumer services can connect to without knowing the actual address of the destination service. The consumers or subscribers subscribe to something called a topic in case of kafka, and are notified each time data is available for processing in the queue. Whenever the consumers are free, they read the message from the queue, read it and then remove it from the queue. Multiple consumers can read the same topic at their own pace. This leads to decoupling of services - such that the system becomes more robust and failure of one of the nodes in any of the consumer groups will not affect the performance of other nodes processing the data.

- **Fault Tolerance**
  To ensure that messages are delivered even in case of failures of worker nodes, there are three types of options for delivering messages in a broker:-

  1) **Exactly once -** This is the best case scenario, where messages are delivered to the queue exactly once, and are removed from the queue once a worker starts processing the message. The successful delivery of messages can be ensured by receiving acknowledgement from the client. In case the acknowledgement is not received, the message is redelivered to queue for processing after certain visibility timeout. However, this method faces the **"Two General Problems"** issue where it is difficult to guarantee that either messages or acknowledgements will be delivered due to lossy networks.

  2) **At Most once -** In this type, the messages are delivered at most once. In case no acknowledgement is received due to lossy networks, the messages are not delivered to the queue again. This is mostly used in financial systems where duplicate messages pose a threat to system integrity.

  3) **At Least once -** This type of message delivery protocol attempts to deliver messages to the queue until a successful receipt is acknowledged. It is used when message delivery is more important than order in which messages are received by the client.

  As for the implementation of the message queuing system for this project, at least once delivery system will be used. This will ensure that in case of failure of either of the servers or the worker groups, messages will still be processed by the remaining worker nodes, and will be delivered to the client at least once. A copy of message will be replicated across all partitions within the worker group, for each topic to ensure smooth processing of messages. Furthermore, to avoid single-point of failures in case the master node fails, leader election will be implemented to appoint a new leader and this information will then be propagated across different worker groups in different partitions.

- **Scalability**
  The scalability necessitates that the user message load is distributed across servers and represents a departure from previous metadata-private systems (Tyagi, N., Gilad, Y., Leung, D., Zaharia, M., & Zeldovich, N., 2017).The messages are stored in topics which in turn are divided into partitions. As partitions form independent units within a topic, these allow brokers to scale out horizontally and manage load within the cluster. The Cluster capacity can be increased by adding new message queues and partitions. This idea is from Kafka which implements topics and partitions as a way of managing and scaling messages delivery.

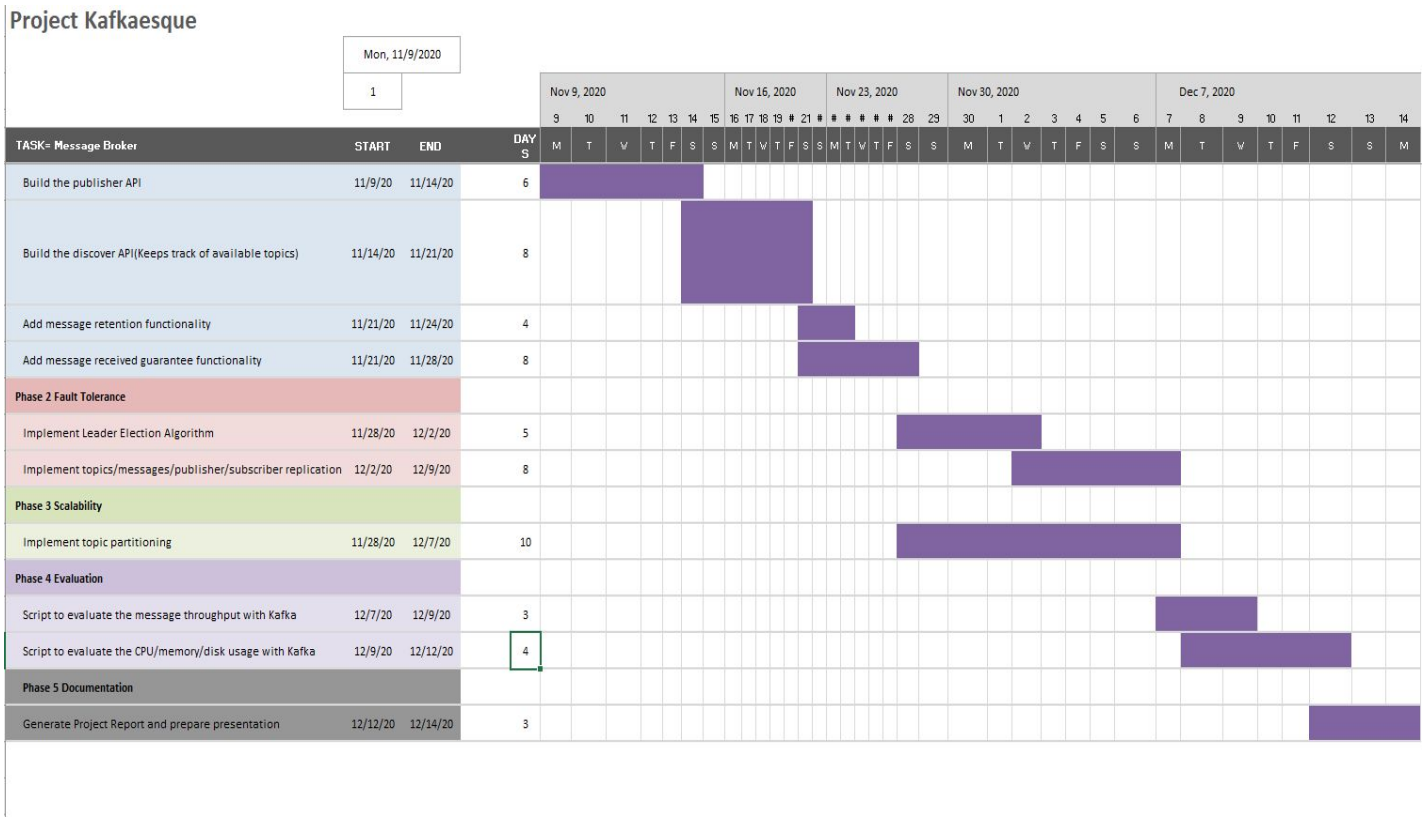
## Project Schedule:

The project will be divided into 5 phases: -

**Phase 1: Broker implementation -** Implement the publishers, subscribers and broker modules
**Phase 2: Fault tolerance -** Implement leader elections and replication of messages for fault tolerance
**Phase 3: Scalability -** Implement topic partitioning
**Phase 4: Evaluation -** Evaluate the distributed system against Kafka
**Phase 5: Documentation -** Document the project as a report and prepare presentation

The details related to each phase as well as the timeline for the implementation details are displayed in the Gnatt chart shown below, and also can be found on this link: -

https://github.com/gwDistSys20/project-kafkaesque/blob/main/milestone-3/Project%20Kafkaesque.xlsx




## References:


Kreps, J., Narkhede, N., & Rao, J. (2011, June). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (Vol. 11, pp. 1-7).

Tyagi, N., Gilad, Y., Leung, D., Zaharia, M., & Zeldovich, N. (2017, October). Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (pp. 423-440).

Intorruk, S., & Numnonda, T. (2019, July). A Comparative Study on Performance and Resource Utilization of Real-time Distributed Messaging Systems for Big Data. In *2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* (pp. 102-107). IEEE.

Sharvari, T., & Sowmya Nag K. (2019). A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming. *CoRR*.

Aung, T., Min, H. Y., & Maw, A. H. (2018, October). Performance Evaluation for Real-Time Messaging System in Big Data Pipeline Architecture. In *2018 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)* (pp. 198-1986). IEEE.

Aung, T., Min, H. Y., & Maw, A. H. (2019, November). Coordinate Checkpoint Mechanism on Real-Time Messaging System in Kafka Pipeline Architecture. In *2019 International Conference on Advanced Information Technologies (ICAIT)* (pp. 37-42). IEEE.

Dobbelaere, P., & Sheykh Esmaili, K. (2019). Industry Paper: Kafka versus RabbitMQ. *A comparative study of two industry reference publish/subscribe implementations*.

Vanburgh, G. (2016). Writing a Message Broker in GoLang. Retrieved 2020, from http://studentnet.cs.manchester.ac.uk/resources/library/3rd-year-projects/2016/george.vanburgh.pdf