# 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 emulates a publish subscribe communication model for distributed systems for collecting and distributing real-time data generated from multiple sources using existing systems such as Apache Kafka and RabbitMQ as a reference and making improvements over them. As a part of implementation, a publisher-subscriber model for distributing messages is created in which multiple consumers (VM's, servers, etc.) form consumer groups to subscribe to a topic to receive data from a message queue known as broker on which the producers publish data on different topics. Some of the advantages of implementing a publisher-subscriber model is it aims at scalability and also addresses an old messaging problem called group messaging. This project also explores how to partition the log messages across topics and replicate them across these partitions to achieve fault tolerance.

## Related Work

There are two models by which we can build a dynamic distributed messaging system. Those are:

1. Publish/Subscribe Model.
2. Messaging Queue Model.

**1. Publish/Subscribe Model:**

It is a mechanism where the publisher posts the data/streams of data to the server. The data is grouped into various categories/classes. The subscriber/consumer subscribes to various classes and thus retrieves the data. The broker makes sure that the subscriber subscribed to a particular class/category of data will receive data at any time. Popular distributed messaging systems which rely on the publish/subscribe paradigm are Apache Kafka, Apache pulsar etc. The core features of publish/subscribe paradigm are:

1. Time Decoupling.
2. Space Decoupling.
3. Synchronization Decoupling.

**Apache Kafka:**

This design was introduced by LinkedIn. It is based on the publish/subscribe model written in Scala language. It consists of consumers, brokers, topics, clusters and producers. The producers can publish messages to a topic. A topic is a stream of messages categorized by a particular type. Next, the topic is posted to one or more servers often called as brokers. This kind of model is called a push-based model where the streams of a message are pushed on one of the available brokers. A consumer can retrieve a stream of messages from the broker where the message is stored. This kind of model is called a pull-based model where the message is retrieved from the broker. The broker does not have any state i.e. the broker does not keep track of how many consumers consumed the information, which information is retrieved etc. Apache kafka is distributed in nature and therefore it has multiple brokers which can store the data/stream of messages. Zookeeper maintains all the services that are required to maintain the communication between the producer and consumer(Aung, T., Min, H. Y., & Maw, A. H. ,2018). The whole event can be done either synchronously or asynchronously.

Below is the sample algorithm for producer and consumer as per the authors (Kreps, J., Narkhede, N., & Rao, J., 2011).

**Sample producer code:**

```
producer = new Producer(...);
message = new Message(“test message str”.getBytes());
set = new MessageSet(message);
producer.send(“topic1”, set);
```

**Sample consumer code:**

```
streams[] = Consumer.createMessageStreams(“topic1”, 1)
for (message : streams[0])
{
bytes = message.payload();
 // do something with the bytes
}
```

**Apache pulsar:**

Apache pulsar is one of the popular distributed messaging systems. It was developed by yahoo in 2014. Pulsar is horizontally scalable. A pulsar cluster handles the bandwidth of messages between producer and consumer. Pulsar uses zookeeper and in addition to that it also uses bookkeeper (Intorruk, S., & Numnonda, T. 2019). Pulsar provides end-end latency.

**2. Messaging Queue Model:**

It is majorly used for inter-process communication. In this kind of architecture, the publisher delivers the message to a message queue. If a consumer wants to consume the message, the consumer connects to the message queue and retrieves the message. The messages are kept live on the message queue until a consumer consumes the message. Each Message is processed only once. The entire event is done in an asynchronous way i.e. producer and consumer are not required to interact simultaneously with the message queue. Messaging systems like RabbitMQ, IBM MQ are based on the messaging queue paradigm.

**RabbitMQ:**

RabbitMQ is an open-sourced framework that implements Advanced Messaging Queuing Protocol(AMQP). It enables asynchronous message based communication, by enabling loose coupling between server and clients in a cluster, i.e. they need not run at same time. Additionally, this framework can be implemented in any language because of its language agonistic nature. In RabbitMQ, the messages are transported over TCP connections (Sharvari, T., & Sowmya Nag K., 2019). It consists of a Publisher, Consumers, Exchange and Queues. Every message contains a payload containing the message and a routine key which determines the queue in which the message will be placed.

**NATS streaming model:**

Additionally, there are some frameworks such as NATS Streaming which support both the publish/subscribe and message queuing systems. This is a lightweight, open-source, cloud-based messaging service implemented in Golang and is maintained by Synadia Group. In the publish/subscribe based model, publishers send the message to the NAT subjects, which are subscribed by the consumers. The consumers listen for any data published to these NAT subjects in order to receive messages. Scalability and

fault tolerance in case of node failures are handled by a master server known as "**gnatsd server**" by cutting off subscriptions that extend a certain timeout value.

The NATS server in turn is embedded within NATS streaming service, which provides an API to communicate with NATS server. The go channels act as subjects in this case, where producers publish messages and subscribers receive messages. This service guarantees at least once delivery of messages using google protocol buffers as well as maintains persistence of messages using durable subscriptions, where in case of network failures, the server starts again with the earliest message unsubscribed by the consumer.

## Project Design

This project aims to build a simple publish-subscribe model 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. In this project, the messages are published on the same machine by making telnet calls on different ports on the localhost to emulate the multiple servers publishing messages to the broker. These messages are stored in the local file system, which has the structure '**\broker\topic\file'**, where the broker forms the root folders, which stores the incoming log messages in files across different subfolders for the same topic based on the total number of partitions. While sending messages, publishers only know the topic to which data will be sent to, and are unconcerned about how the subscribers that will consume these messages from these topics. 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 in the local file storage identified by the broker-id, where publishers send the messages in the broker. These 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 messages.

**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. In this project, the messages are sent in the JSON format consisting of header, key and body such as

```
"{""Header"":""header"",""Key"":""124"",""Body"":""Hello World!""}".
```

The header and key form optional fields, where the key field is hashed to determine the partition folder for the current topic to which the message will be published. In absence of key, the log messages are written to the partitions in a round-robin format.

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 topic folder for a broker, they are indexed by an offset stored in the index file, which points to the line containing log data in the log file.

**4) Message Brokers/Queues: -** 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 and helps to decouple sending and receiving of 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.

This version of the message queuing system uses multiple brokers to handle scaling of messages across partitions and to handle single-point of failures in case the leader broker fails. For this, the Raft Consensus Algorithm for leader election and log replication has been implemented, which will appoint a new broker to serve as leader and continue sending and receiving messages between publishers and subscribers, as well as replicate latest log data across topics in each broker, to handle fault tolerance issues. The leader broker will keep track of all the active brokers and handle log replication to followers using the state machine replication algorithm of Raft.

**Interaction between the components: -**

The client acts as a producer and places the request to process the message in a broker using the command **“PUBS topic_name msg”** (Ref: Fig-1.0(a)). The topic_name identifies a topic which 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 once delivery of messages is guaranteed by replicating the logs across multiple brokers, which ensures high availability of data.

The topics are then divided into partitions. The subscribers will subscribe to a topic using the command **“SUBS topic_name consumer_group [no. of partitions to subscribe to]”** by specifying the the topic name from which messages are to be read, consumer group to which they belong and optionally, total number of of partitions in which the consumers will be distributed and receive a subscription-id. Finally, they consume the messages using the command “**CONS topic_name subscription_id offset_value”** by specifying the topic name, subscription-id and the offset value in the index file, which points to the line containing the original log message intended to be read from the log file.

These partitions are in turn replicated to multiple brokers (message queues) to ensure fault tolerance using Raft’s state machine replication mechanism. The replication factor indicates how many copies of partition are maintained for each topic which is entered as a command-line argument by the consumer at the run-time. 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 a single point of failure, the leader election algorithm of Raft is implemented, to appoint one of the available brokers as the leaders. The leader then forwards this information to other brokers, as well as communicates with them to store log data across the partitions in these brokers as well.

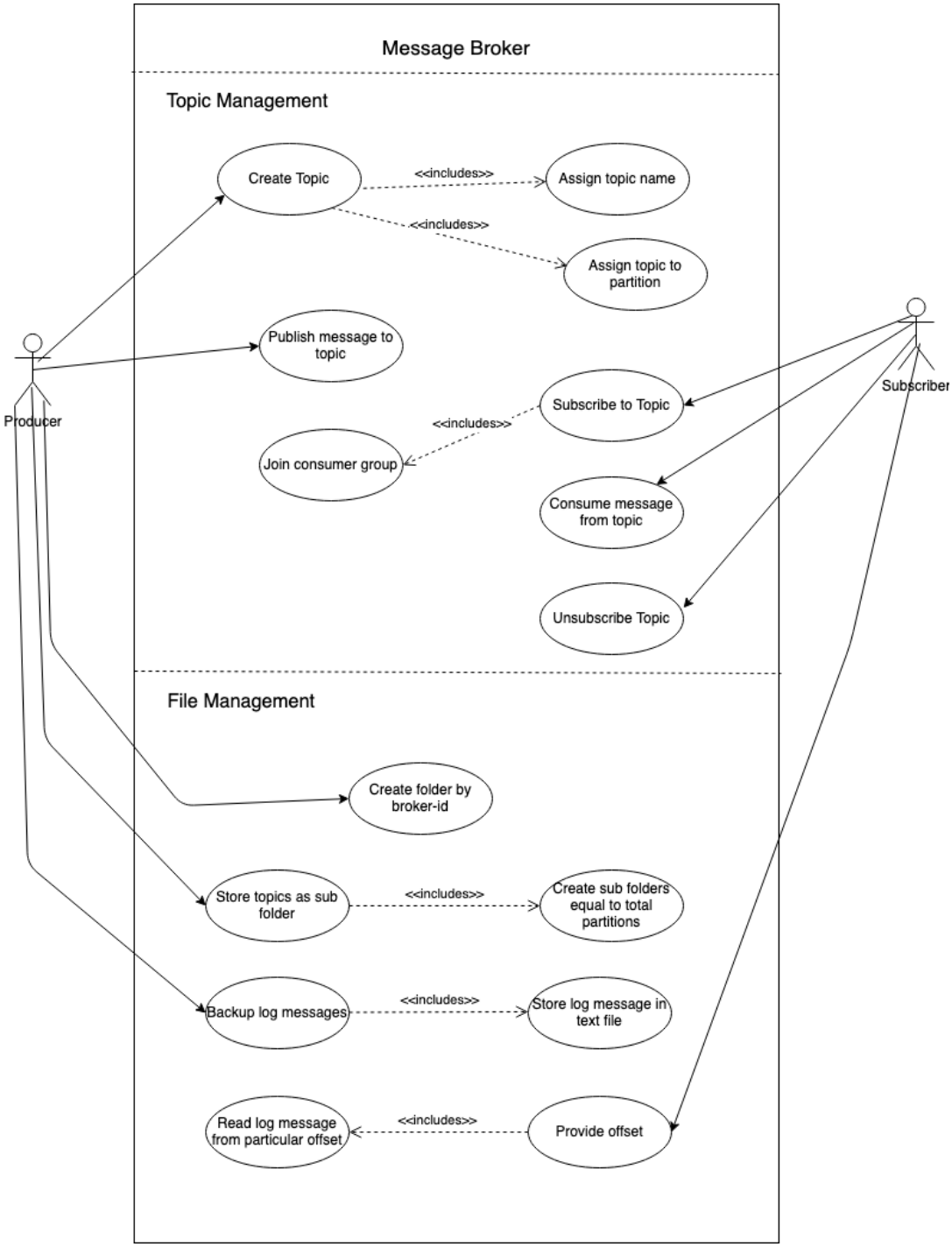


**Fig-1.0 (a)- Use case diagram**

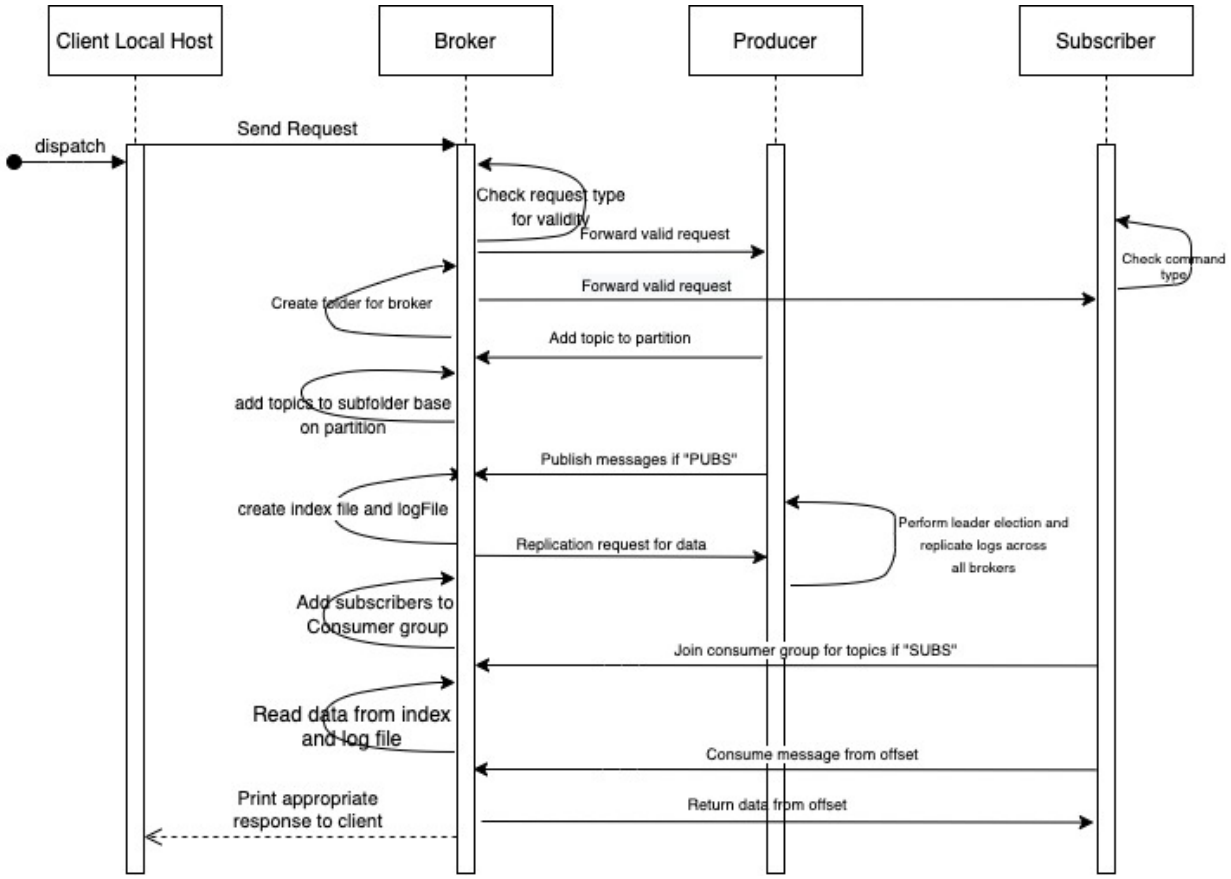


**Fig-1.0 (b)- Sequence diagram**

# Distributed Systems Challenges

## 1. Heterogeneity

The distributed system contains many different kinds of hardware and software working together in cooperative fashion to solve problems. Heterogeneity in computing generally refers to different Instruction Set Architectures (ISA) in multiple processor environments, but may also include different platforms such as heterogeneous Operating Systems, networking devices, and programming languages.

**Relation with project-Kafkaesque:-** This implementation is based on single hardware and single operating systems as the application runs on the localhost and communicates with the server by making telnet connections. To make a real-time application, the system could be enhanced further in future to connect

to different Virtual Machines running on cloud platforms such as Amazon Web Services, Microsoft Azure or Google Cloud Platform (GCP).

## 2. Openness

This distributed systems challenge deals with the standards that should be set in order for the system to be extended and reimplemented by other developers in different manners. It essentially measures the degree to which new resource-sharing services can be added by someone not involved in the development team of the project to the client-program. A well-defined Application Programming Interface can aid the development of a project, by adding/removing features by the developers after releasing the application for use in public. Most of the open-source projects supported by The Linux foundation such as NodeJS API and the Linux operating system fall address this challenge.

**Relation with project-Kafkaesque:-** The openness challenge is addressed by this system in the form of ability to integrate our project easily with other applications by developing libraries written in their preferred language.

## 3. Security

Security is one of the important challenges in distributed systems. Due to various features available in a distributed system, it is difficult to manage the security policies of the system. Additionally, this challenge conflicts with the Openness challenge of the distributed system, and the more open a system is, the attackers have a higher chance of finding a vulnerability for penetrating the system.
**Relation with project-Kafkaesque:-** Security has three components: Confidentiality (who is authorized to access data), integrity (the original data should not be modified by anyone unauthorized to access the data) and availability (the data should always be available to authorized entities for access ). As of now, our project is partially secured and addresses the availability and integrity issue. As if exploited by an attacker, the original data related to the topics is stored in the backup file system, and can be retrieved easily. This feature can be further developed by providing a two factor authentication, while connecting to the broker server and encrypting all the data before passing through the network and storing it in files, for a real-time system.

## 4. Failure Tolerance

A program may return incorrect results or may crash before completing intended computation, due to faults in either hardware or software. The failures in a distributed system may be either crash failures resulting from abrupt shutdown of

a server, or malicious failures resulting from a node in the cluster being compromised. A good distributed system is one which continues execution and returns correct results, by ensuring that failure of one component should not stop others from execution.

**Relation with project-Kafkaesque:-** It is hard to detect failures in distributed systems as some nodes assumed to be crashed, might be slow in executing the commands and might have not crashed at all. This system implements a Raft Consensus Algorithm which takes care of replicating data on multiple brokers as well as performs leader election for brokers to avoid single-point of failures, in case leader broker crashes. Apart from this, the data from brokers is replicated across partitions which form folders in the backup file system and stored into the files using a file manager script which makes our system less prone to data loss.

## 5. Concurrency

In a distributed system, there may be some resources which are shared between the servers of a cluster. As the distributed systems are designed to execute multiple servers concurrently and in parallel, there is a possibility that many threads or processes may try to access these shared resources at the same time. For instance, a banking database accessed by two processes belonging to different regions at the same time, may have issues of lost updates, if both the processes update the transaction value at same time. Hence, it is necessary to synchronize the access to the shared variables to ensure data consistency.

**Relation with project-Kafkaesque:-** The standard technique to achieve synchronized access used by most operating systems, includes variables known as semaphores and mutexes. This project achieves concurrency by letting multiple producers to send messages to the same topic(s) and multiple consumers consuming messages from the same topic(s) and each of the server runs in a separate goroutine. The synchronization is achieved by using sync.mutex package of golang, by using Locks which ensure that only one producer/consumer process can access the broker server for publishing or consuming topic at a time.

## 6. Quality of Service

Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The quality of a service is usually measured in terms of response time (time taken to process the request for the first time after submission), turnaround time (total time taken to process the requests and return

results), network latencies (delay caused in sending request/response) and throughput (total number of executions per unit time). The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are *reliability*, *security* and *performance*.

**Relation with project-Kafkaesque:-** The current implementation works well when tested on the local system for three broker servers, and performs all the operations such as publishing and consuming message from topics within few milliseconds. The project is designed to run as many commands as entered from telnet command line by client, until terminating escape character is entered. The failures are handled internally using raft algorithm, to ensure constant availability of data to clients. This could be further enhanced by deploying the project on virtual machines provided by cloud provides like Azure and AWS, and monitoring the system metrics.

## 7. Scalability

Scalability is defined as the ability to increase/decrease the number of resources on demand to run the entire system smoothly.

**Relation with project-Kafkaesque:-** In this project scalability is one of the most important implementations from distributed system challenges. The messages are filtered by topics which are in turn divided by the total number of 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 replicating partitions across multiple brokers, which is a concept referenced from Kafka which implements topics and partitions as a way of managing and scaling messages delivery. Finally, the incoming requests are load-balanced between partitions using a round-robin algorithm.

## 8. Transparency

Transparency is the abstraction of low-lying layers of the multi-tier and multi-layer applications, and conceal how the components of the distributed system are separated from each other to the end-user, so that the system is perceived as a whole rather than as a collection of independent components. In other words, this issue deals with hiding the complexity in design of the system from the end-user as much as possible. The end-user should not be aware of failures happening inside the system due to node crashes, and should get the intended results as requested from the system.

**Relation with project-Kafkaesque:-** This project achieves transparency by abstracting the internal mechanism of how messages are sent to different topics and by the publishers and how they are consumed by the subscribers. The end client has no information of the failures of broker, as the multiple brokers run at same time, and handle leader failures by performing leader election.While scaling the application, that is, adding partitions to the topics, the user still gets the same experience as earlier without knowing that scaling is being done, as the topics are replicated across the partitions and the all the partitions, with topics and log messages are stored in the backup file system.

## Project Outcomes and Reflection

In this project, we successfully implemented a distributed messaging system which emulates a publish-subscribe model. We also dealt with some distributed system challenges like fault tolerance, scalability, openness and transparency. The key difference and selling point of current implementation as compared to existing systems is that it implements raft algorithm at both broker and partition level which gives enhanced scalability and fault-tolerance. The messages are published to a partition by hashing a key, which can be provided either by client at runtime or using a round-robin algorithm. Finally, a file manager system ensures that backup of all the log messages is replicated across the topics in all the brokers to handle node failures in the system.

As the project was not tested against metrics, it is hard to provide actual numbers on how the system performs as compared to existing implementations in terms of throughput and latency. However, in future, this project can be deployed across multiple Virtual Machines across multiple regions using cloud providers like AWS, Microsoft Azure and Google Cloud Platform. The metrics provided by these systems can be used to see how the system works in terms of response time, throughput, and network latencies, and could be a real-time distributed system.

Another issue with the current system is that the traditional raft algorithm at partition granularity has a lot of chattiness as each partition is now communicating with all its replicas. It would be interesting to see if we can further improve our system by implementing it on the MultiRaft algorithm applied by CockroachDB and solve the issues of Raft algorithm. Finally, we are trying to add a dynamic service discovery to discover broker nodes at runtime using Serf library, instead of hardcoding them as peers as in the current implementation.

## 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

Bouchrika, I. and Bouchrika, I., 2020. *Challenges For A Distributed System*. [online] Ejbtutorial.com. Available at: <https://www.ejbtutorial.com/distributed-systems/challenges-for-a-distributed-system> [Accessed 17 December 2020].