International Journal for Research in Engineering Application & Management (IJREAM)
ISSN : 2454-9150 Vol-07, Issue-07, OCT 2021

A Study on Messaging Systems- Kafka, KubeMQ, IBMMQ and RabbitMQ

Rujula Singh R, Computer Science Student, RV College of Engineering, Bangalore India, rujulasinghr.cs17@rvce.edu.in

Nikhil S Nayak, Computer Science Student, RV College of Engineering, Bangalore India, nikhilshivakn.cs17@rvce.edu.in

Abstract A messaging system is responsible for movement of data from one application to another so the application can focus on data without wasting time on data transmission and sharing. Message systems play a vital role in data driven application with huge amount of data needs to be transferred between the applications. The crux of many big data streaming applications, cloud native based applications and microservices-based architecture is distributed messaging system. In the era of time critical applications and real time based applications there is a need for highly efficient, fault tolerant with graceful degradation, scalable and low latency messaging systems. Different applications require different features of the messaging system therefore an in-depth study of popular messaging systems is necessary. This survey paper dives deep into state of art messaging systems-Apache Kafka, KubeMQ, RabbitMQ and IBMMQ. This is to help users decide among many messaging systems. We discuss similarities, use cases and differences. It is of seminal importance for future development and research.

Keywords — IBMMQ, Kafka, KubeMQ, Messaging Systems, Publish-Subscribe, RabbitMQ.

I. INTRODUCTION

Data pipelines help you migrate data from wherever it has been created to various other places where there is a possibility of processing it. This is a central idea of a data-driven application. In messaging systems data consumption and data transmission are two different processes, that is, it's not carried out by the same process as, there is no direct connection between a sender and a receiver. This is a smart concept as it prevents applications from wasting time on consuming and transmitting data. Messaging systems should be able to improve performance, scalability, manageability, and reliability. Messaging is a wide term that encompasses a variety of methods each of which differ in terms of how this data is transferred from sender to receiver. Publish-subscribe and message queuing are the two main types of messaging models.

A. Publish/Subscribe

Publish subscribe model also popularly known as a pub/sub model is an asynchronous communication between services. This model is not only popular with serverless but also microservices based architecture. Immediate responses to the receivers are obtained when the corresponding topic has been populated by the publisher. If your application is event-driven or decoupling is of cardinal importance, then pub/sub messaging can boost performance reliability and scalability.

The Publish/Subscribe messaging pattern brings many benefits like Implementing the publisher and subscriber parties independently from each other, publishers and subscribers aren't required to know each other, one Subscriber could receive data from many different Publishers and a single Publisher could send data to many different Subscribers. Figure 1 shows the pattern of a pub/sub, where we have a single publisher and many subscribers for a given channel.

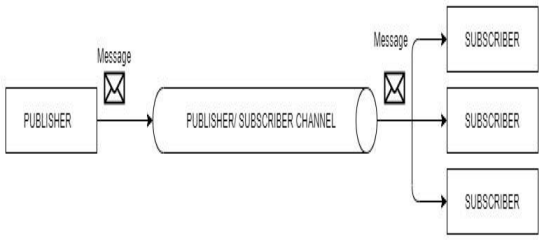

Fig. 1. Pub/Sub Pattern

B. Messaging Queue Paradigm

In serverless and microservices systems, a message queue is a type of asynchronous service-to-service communication. Messages are queued until they are processed and then discarded. A single consumer processes each message just once. Message queues are useful for decoupling heavy processing, buffering, or batching work, and smoothing spiky workloads. This paradigm is useful when we want to

 DOI : 10.35291/2454-9150.2021.0577

converse via short message formats. When many processes communicate at the same time, shared memory data must be secured via synchronization. If the frequency of writing and reading from shared memory is high, then implementing the functionality will be difficult. What if most processes don't need to access shared memory, but only a few do? In all the above scenarios message queues would be a preferable solution. The first message inserted in the queue is the first one to be retrieved. (FIFO)

II. Related Work

This section recapitulates previous research on message queuing systems. In [2], [3] the authors have qualitatively distinguished the important differences within message queuing systems. [2] compared system usage among many other things which includes download, installation, and documentation. [3] compares the communication and design of many message queuing systems. [5] and [6] investigate the performance differences between Kafka and RabbitMQ. [1], [7] undertake a quantitative and qualitative review, although their experimental results are based on each system's built-in test tools, which may cause fairness difficulties other resources have compared KubeMQ and IBMMQ based on portability, configuration requirements, deployment time, availability, payload size, dependencies on external systems.

The majority of existing research has concentrated on at the most two to three message queuing systems. After an in-depth literature survey, it was found that there is no comprehensive study that compares the four prominent systems on multiple dimensions. Furthermore, while a quantitative comparison is carried out for the systems being tested, the experimental findings are produced utilizing the systems' built-in test tools. Therefore, this produces biased results. We try to bridge the gap by using a general test framework which provides an unprejudiced comparison. We also go over the best use cases for the four different systems under consideration, which will aid users in selecting the right message queuing system for their needs

III. Overview

In this section we discuss in detail about messaging systems that is, publication-subscribe model or message queuing model. It also helps understand prominent features of four of the most pervasive messaging systems.

A. *Apache Kafka*

Apache Kafka is an open-source software platform. It is written in Java and Scala. It is a product of Apache Software Foundation. It has many applications in streaming analytics and integrating data pipelines. It is also useful for mission critical applications. This distributed system consists of multiple servers and clients. Communication between them is facilitated through a TCP network which is highly efficient. It comprises of Producers and Consumers. Producers write data into topics whereas consumers read data from these topics. Each and every topic has a log associated with it that exists on a disk in the form of a data structure. Records are appended to a topic log which is generally added at the end of file without replacing existing data usually from single or multiple producers in Kafka. A single topic can have many partitions. Each partition is not necessarily stored on the same node. Consumers can read or consume data from any topic log and can pick up where they have left off. Consumers belong to a particular consumer group. Each group maintains offsets to enable consumption of new data. Kafka provides high performance and horizontal scalability by storing data on different nodes. Kafka provides two models - one is message queuing and the other is publish subscribe. Kafka messaging queue has a single server that can act as a source for a pool of consumers. Each record is only sent to a single consumer to prevent duplication. In publish subscribe, the record is available to all consumers that is, the data is broadcasted for everyone to consume. Figure 2. shows the high-level architecture of Kafka with a cluster, producer and consumer group coordinated with the help of zookeeper.

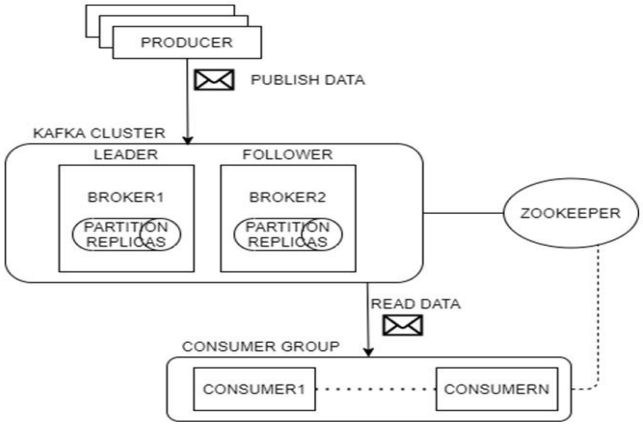

Fig. 2. Kafka Architecture

B. *RabbitMQ*

RabbitMQ is an open-source software. It is written in Erlang. RabbitMQ like many other messaging systems uses a cluster-based approach with usually a single broker. RabbitMQ is a highly available and flexible system. Effective queuing is achieved with the help of a pliable routing system. Customize routing is available for convoluted operations. It is also available in other powerful languages apart from Erlang and multiple protocols can be utilized.

RabbitMQ can be implemented using many protocols. AMQP 0-9-1 is a protocol with two states, and messaging semantics that are far from tenuous. It is a developer friendly protocol and has robust client libraries which are adaptable to a variegated number of programming languages and environments. STOMP is a basic protocol

 DOI : 10.35291/2454-9150.2021.0577

which underscores simplicity as it uses text messages. It can also leverage protocols that are built on top of telnet. MQTT similarly is a protocol with two states that underscores the importance of high speed publish / subscribe messaging, popular among clients using highly constrained devices. However, it is only well suited for publish/subscribe systems. HTTP can also be used as the messaging protocol.

RabbitMQ uses modifications of request/reply, publish/subscribe and point to point messaging patterns. It utilizes a Fast broker/slow consumer model — that provides a reliable delivery of messages while simultaneously monitoring the consumer state. It is adaptable to popular languages such as Java, .NET, Scala, Ruby and node.js. It can leverage synchronous or asynchronous distributed systems. It is an implementation of the Advanced Message Queuing Protocol (AMQP). With this type of message model, the producer, in our case, checks out the service that produces the messages. Instead of producing directly to a message queue, it's going to produce to exchange. We can think of the exchange as a post office; it's going to receive all messages and then distribute them according to how they are addressed. Figure 3 shows high level architecture of RabbitMQ with three brokers in a mirror cluster enabling communication between multiple producers and consumers.

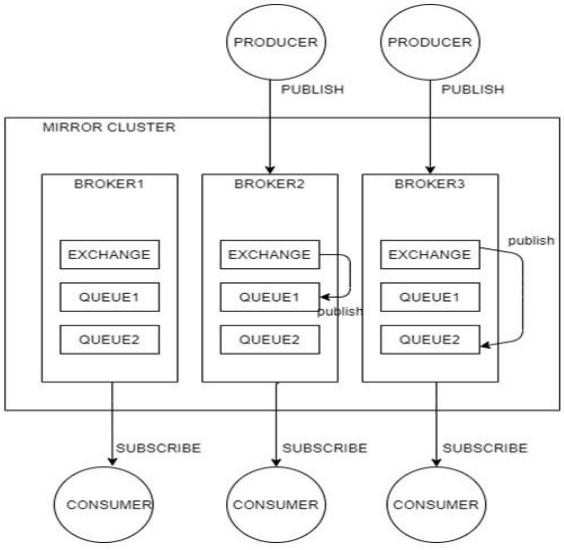

Fig. 3. RabbitMQ Architecture

C. KubeMQ

It is written in GO. Kubernetes is central to KubeMQ. It is a high-quality message broker with plug-and-play connectors and distributed monitoring. It enables transparency in private as well as public clouds, with easy integration to microservices. It is a modern messaging queue with a high-speed broker, certified in the Cloud Native Computing Foundation ecosystem. These systems are easily deployable in Kubernetes cluster which can have a down time of less than one minute. DevOps operations are largely simplified with the help of client friendly SDKs and libraries. The four messaging themes used by KubeMQ are - Message Queuing, Streaming, Publish/Subscribe, and Remote Procedure Calls. It is a viable solution for a myriad of use cases because of its flexibility. It's hybrid infrastructure provides a solution that is robust to many different types of environments across public and private clouds.

D. IBMMQ

It provides two alternative application programming interfaces (APIs) for use in Java applications: classes for JMS as well as for microservice based Java applications. It is highly popular among leading organizations and has a good reputation in the developer community. IBMMQ has the largest market presence among products in Message Queuing Systems. IBMMQ architectures range from simple architectures using a single queue manager, to more complex networks of interconnected queue managers. Multiple queue managers are connected together using distributed queuing techniques. IBMMQ provides two different release models The Long-Term Support (LTS) release is most suitable for systems requiring a long-term deployment and maximum stability and The Continuous Delivery (CD) release is intended for systems which need to rapidly exploit the latest functional enhancements for IBMMQ. The four messaging themes used by IBMMQ are - Message Queuing, Streaming, Publish/Subscribe and File transfer. Applications can also broadcast messages. Figure 4 shows a high-level architecture of IBMMQ with two queue managers in a cluster enabling communication between multiple publishers and subscribers.

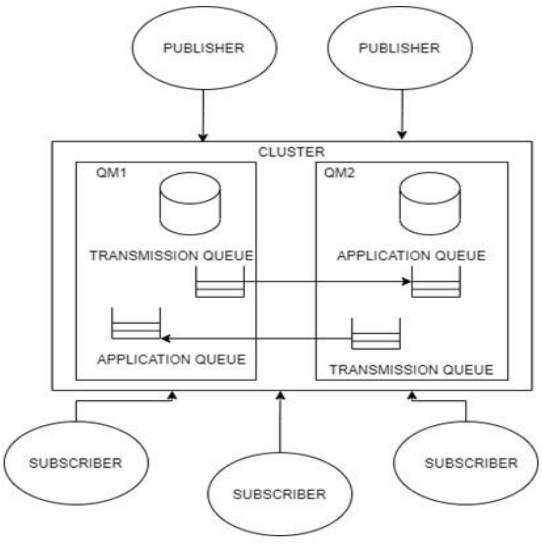

Fig. 4. IBMMQ Architecture

IV. Feature Comparison

The four technologies mentioned above are all popular, and they have so many similarities that choosing the correct framework might be challenging. Here, the Quality of Service (QOS) is investigated in-depth for the different messaging systems mentioned above. This will help

 DOI : 10.35291/2454-9150.2021.0577

determine the extent of scalability and quality of the system and its ability to fulfill the requirements of the customer. On many QoS criteria, a full comparison is done between all four. Table-1 shows the comparison on popular features.

A. *Message Delivery*

The guarantee of message delivery is at the heart of service excellence. The delivery method known as 'at most once' is one in which the message may or may not be delivered resulting in high throughput. 'Exact once' delivery is when the message is received by the consumer only once. This necessitates time-consuming calculations. When a message is sent at least once, but it can possibly be delivered several times, it is referred to as 'at least once' delivery. This is helpful in the event of a failure. Multiple types of delivery like 'at-most once', 'exactly once' and 'at-least once' delivery is provided by Kafka [10]. Similarly, 'at most once' and 'at least once' delivery is provided by RabbitMQ. KubeMQ supports asynchronous and Synchronous messaging with support for guaranteed message delivery, 'At Most Once' Delivery and 'At Least Once' Delivery models. IBMMQ provides 'once and once only' delivery of messages to ensure messages will withstand application and system outages.

B. *Message Persistence*

It is the capacity to save messages so that they can be retrieved even if the broker is restarted. In the case of Kafka, the log can be saved on disc and a message retention period can be configure Persistence is an option in RabbitMQ, and it can be saved in memory or on disc. [12]. Message persistence is not guaranteed even if a queue is set to be persistent in RabbitMQ. KubeMQ provides customizable features to configure the expiry time and delay of messages at granular level of a single message. Also, unprocessed messages are added to a dead-letter queue for debugging and logging purposes. In IBMMQ, when queue manager recovers from a failure the messages are persisted as necessary from the dead-letter queue. However, if the messages are not recorded in the dead-letter queue, either due to failure of the system or because of a command issued by the user, the messages are lost without being persisted.

C. *Message Ordering*

Within a partition, there is ordering in Kafka. For worldwide ordering, high cost configurations must be set up which lowers performance. KubeMQ follows FIFO (First In First Out) ordering of messages in case of durable queues. IBMMQ follows FIFO ordering of messages but also maintains priority within the queue of messages.

D. *Latency*

Latency is defined as delay incurred in a process. Kafka and RabbitMQ generally provide very low latency. The authors in [10], [15] observed that in Kafka's 'at least once' delivery pattern, for a medium load the latency doubles, whereas in RabbitMQ's 'at most once' and 'at least once' delivery there is no noticeable change in the latency. Accessing data from the disc would further increase the latency in Kafka. KubeMQ supports high volume messaging with low latency [11]. IBMMQ also supports low latency.

E. *Availability*

It is a system's ability to maximise its uptime. For high availability, the system must provide fault tolerance. Experiments on queue replication in RabbitMQ are undertaken in to investigate the impact on availability and scalability. It has been noticed that performance is best for a single queue, and that when the queue is duplicated, the performance takes a hit, but the upside of this trade-off is increase in fault tolerance of the system. Replication to increase reliability is characteristic of Kafka which is configurable by the developer. The replicated data is stored on multiple brokers. This also has the added advantage of increased fault tolerance. Kafka uses the hadoop ecosystem resource manager in zookeeper [17], which manages the communication between all the available brokers, producers and consumers to coordinate the working of the entire system. Replication also means highly available systems which further provide better performance. High availability translates to critical reliability which enables KubeMQ to transfer large amounts of data without any hassle. If you want to operate your IBMMQ queue managers in a high availability (HA) configuration, you can set up your queue managers to work either with a high availability manager, such as PowerHA for AIX or the Microsoft Cluster Service (MSCS), or with IBM MQ multi-instance queue managers. On Linux systems, you can also deploy replicated data queue managers (RDQMs), which use a quorum-based group to provide high availability.

F. *Scalability*

It describes a system's ability to adapt to a rising number of producers, consumers, or brokers. RabbitMQ enables clustering, which allows several nodes to function as a single message broker. This is beneficial for balancing demand and scaling the system to handle a huge number of messages. Kafka was created as a horizontal scaling system from the bottom up. Kafka is easy to grow thanks to the zookeeper's coordination in adding and deleting brokers. KubeMQ is adopted for seamless container management; it provides high scalability and enhances communication or messaging. It can also allow for the addition of many lifetime applications when building a microservice. In IBMMQ, by adding instances of service queues we can start to scale applications. IBMMQ Clustering allows instances to be added or removed without modifying client or server applications.

 DOI : 10.35291/2454-9150.2021.0577

V. DISCUSSION

Section 4 delves into the feature comparisons in depth. Table-1 does a comparison of the four messaging systems with respect to some features. The section also includes a list of common features shared by the four systems, which can be studied further by comparing them to other messaging frameworks. While coming up with a design solution or establishing a distributed architecture, the important considerations were elaborated in the previous sections.

Feature	Messaging Systems			
	Kafka	*KubeMQ*	*RabbitMQ*	*IBMMQ*
Run anywhere	yes	yes	yes	
Persistence	yes	yes	yes	yes
High Availability	yes	yes	yes	yes
Exactly once Delivery		yes	yes	
Message Expiration		yes	yes	yes
Delayed Delivery		yes		
Long Polling		yes	yes	
At most once delivery		yes	yes	
Consumer groups	yes	yes		yes

Table 1. Feature Comparison

When it comes to distributed log systems, Kafka is more mature, and it is the leading solution for real-time data analytics where high throughput and low latency are of cardinal importance. KubeMQ is preferable for relatively simple and high-speed applications. It has a small support network that is continuously expanding as new releases are released. More features are added with each version, which will soon make it competent with Kafka on all fronts. Kafka can be utilized in situations when real-time data is not crucial, such as when the system is not affected even when a small number of messages are lost during transmission. Examples of such situations are: Advertisements on websites, social media and user tracking using cookies. RabbitMQ is a basic message broker that can handle sophisticated routing via exchanges and queues. In order to create intelligent applications with a large number of sensors that are connected to the internet, unique routing is required. For financial transactions where each message is of paramount importance, the acknowledgement of each transaction is useful which is provided by RabbitMQ. The four messaging themes used by IBMMQ are - Message Queuing, Streaming, Publish/Subscribe and File transfer. Using multicast, applications can send messages to a large number of people.

VI. CONCLUSION

This paper explores the publisher/subscriber and messaging queue concepts in detail. The article gave a quick overview of four major messaging frameworks, including Apache Kafka, RabbitMQ, KubeMQ and IBMMQ. Then there were feature comparisons concerning message durability and ordering, throughput, and latency. Also, this paper emphasizes the importance of scalability and availability in distributed messaging systems. These aspects are critical in deciding whether a framework is suitable for a certain application. As a result, the paper provides guidelines on how to choose the most suitable messaging system for different scenarios. Non-compliance with these standards will result in additional charges during the application development process. As more powerful silicon devices with great processing capabilities become available, messaging frameworks will become the backbone of future technologies. Future study will include a deeper dive into the technological advances in the field with emphasis on some of the newer messaging systems.

ACKNOWLEDGMENT

We would like to thank all the faculty of RV College of Engineering and others for their knowledge transfer and the support provided for this work on Survey of Messaging Systems.

REFERENCES

[1] G. Fu, Y. Zhang and G. Yu, "A Fair Comparison of Message Queuing Systems," in IEEE Access, vol. 9, pp. 421-432, 2021, doi: 10.1109/ACCESS.2020.3046503.

[2] B. R. Hiraman, C. V. M and C. Karve Abhijeet, "A Study of Apache Kafka in Big Data Stream Processing," 2018 International

[3] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), 2015, pp. 132-137, doi: 10.1109/RoEduNet.2015.7311982.

[4] M. Y. Afanasev, Y. V. Fedosov, A. A. Krylova and S. A. Shorokhov, "Performance evaluation of the message queue protocols to transfer binary JSON in a distributed CNC system," 2017 IEEE 15th International Conference on Industrial Informatics (INDIN), 2017, pp. 357-362, doi: 10.1109/INDIN.2017.8104798.

[5] John, Vineet & Liu, Xia. A Survey of Distributed Message Broker Queues.2017, arXiv:1704.00411

 DOI : 10.35291/2454-9150.2021.0577

[6] T. Treat. Benchmarking Message Queue Latency, http://bravenewgeek.com/benchmarking-message-queue-latency/,2016

[7] M. Rostanski, K. Grochla and A. Seman, "Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ," 2014 Federated Conference on Computer Science and Information Systems, Warsaw, 2014, pp. 879-884

[8] J. Kreps, N. Narkhede and J. Rao, "Kafka: A distributed messaging system for log processing", Proc. Int. Workshop Netw. Meets Databases, vol. 11, pp. 1-7, 2011.

[9] S. Dixit and M. Madhu, "Distributing messages using Rabbitmq with advanced message exchanges", Int. J. Res. Stud. Comput. Sci. Eng., vol. 6, no. 2, pp. 24-28, 2019.

[10] J. Yongguo, L. Qiang, Q. Changshuai, S. Jian and L. Qianqian, "Message-oriented middleware: A review", Proc. 5th Int. Conf. Big Data Comput. Commun. (BIGCOM), pp. 88-97, Aug. 2019.

[11] John, Vineet & Liu, Xia. A Survey of Distributed Message Broker Queues.2017, arXiv:1704.00411

[12] P. Le Noac'h, A. Costan and L. Bougé, "A performance evaluation of Apache Kafka in support of big data streaming applications," 2017 IEEE International Conf

[13] M. Rostanski, K. Grochla and A. Seman, "Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ," 2014 Federated Conference on Computer Science and Information Systems, Warsaw, 2014, pp. 879-884

[14] S. Skeirik, R. B. Bobba and J. Meseguer, "Formal Analysis of Fault tolerant Group Key Management Using ZooKeeper," 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Delft, 2013, pp. 636-641.

[15] Martin Toshev, “Learning RabbitMQ” Birmingham, UK: Packt Publishing, Ltd, 2015

[16] B. R. Hiraman, C. V. M and C. Karve Abhijeet, "A Study of Apache Kafka in Big Data Stream Processing," 2018 International Conference on Information, Communication, Engineering and Technology (ICICET), Pune, 2018, pp. 1-3.

[17] S. Patro, M. Potey and A. Golhani, "Comparative study of middleware solutions for control and monitoring systems," 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, 2017, pp. 1- 10.

[18] Kamburugamuve, Supun & Fox, Geoffrey. (2016). “Survey of Distributed Stream Processing.” 10.13140/RG.2.1.3856.2968.

[19] Z. Wang et al., "Kafka and Its Using in High-throughput and Reliable Message Distribution," 2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS), Tianjin, 2015, pp. 117-120.

 DOI : 10.35291/2454-9150.2021.0577