Low Latency Message Brokers

Ranjith G Hegde[1], Nagaraja G S[2]

[1,2]Dept. of Computer Science and Engineering, R V College of Engineering, Bengaluru

---***---

Abstract - *Message Brokers form an essential and irreplaceable part of microservices, real time and cloud native applications as they are used to communicate between various application services. They handle variety of data of massive volume in short period of time having a high throughput and low latency. Message Brokers supports replication and distribution of workload which are essential for any big data processing or streaming applications. Currently numerous messaging systems are present in the market, with new ones being proposed ever so often. This luxury of choice creates confusion in the industry on what Message Broker to be chosen for application development. In depth study is needed to decide the choice of Message Broker suitable for an application. This paper discusses available Message Brokers and gives deeper insight on three most popular Message Brokers – Redis, RabbitMQ and Apache Kafka. For the benefit of the reader introduction is given on Message Brokers and their use case. Comparison of these systems should facilitate readers in making knowledgeable decision.*

***Key Words*: Message Brokers, Redis, RabbitMQ, Apache Kafka, Pub/Sub, In-Memory Cache**

1. INTRODUCTION

The growth of world wide web (internet) has led to tremendous generation of data of various types and massive volume of different velocity. The data generated is being used in various sectors of E-Commerce, Retail, Health Care, Manufacturing and Internet of Things. The competition grows fierce day by day for insights that these data contains. As the applications are constantly evolving to meet the requirements of the hour low latency cross platform Message Brokers that can adapt to ever changing dynamic environment are growing in importance. Message Brokers are used for low latency communication various application hence, relieving the application from the burden of communication and focus on task at hand. Traditional messaging systems are incapable of handling the volume of big data [1]. As they suffer from high latency communication through HTTP and use of sockets with inefficient thread creation [2]. A model that is scalable and loosely coupled such as publish/subscribe paradigm [3] is suitable for application that need dynamic and scalable communication. Similarly, message queues and in-memory database are go to in handling Logging, User Tracking, Critical Applications and IOT devices [4]. Message Brokers form the backbone of modern systems used for microservices, big data analytics with streaming and cloud native applications.

1.1 Publish/Subscribe Paradigm

Publish/Subscribe paradigm or popularly called as pubsub paradigm, is used for asynchronous communication between services hence, decoupling them in order to provide more flexibility with increased communication speed, scalability and reliability to the developer. The decoupling takes place as follows:

- **Time Decoupling:** The publisher and subscriber need not be online at the same time. Subscribers are notified when there are new messages which they can consume at their own pace.
- **Space Decoupling:** The publisher and subscriber need not know each other. They are only aware about the topic, publishers publish message to topic and subscribers that have subscribed to that particular topic consume messages.
- **Synchronization Decoupling:** The publisher and subscriber are independent of each other and work asynchronously without needing to wait for completion of the others action.

The pubsub brokers are used for balancing workloads, asynchronous workflows, event notifications, data streaming high load system such as streaming analytics, internet of vehicles, online social networks and stock trading. Publish/Subscribe system are hosted as managed service in Amazon AWS, Microsoft Azure and Google GCP for cloud applications [5].

1.2 Message Queue Paradigm

Message queue is a messaging system that allows applications to exchange data and other information. Critical to the use of message queue in a production system is its ability to scale to meet increasing loads and to continue operating even if failure occurs [6]. They also provide an asynchronous communication, where sender and receiver are decoupled. Queue stores the messages until they are consumed by the receiver. The main difference from the previous paradigm is that communication happens point to point. Popular message queues are Amazon MQ, RocketMQ, RabbitMQ, ActiveMQ, ZeroMQ, and IBM MQ.

1.3 In-Memory Cache

With the development in memory technology the cost high speed memory especially cache decreases. Caches have exceptionally high access rate per sec than hard disk and RAMs providing a high throughput and very small response time ideal for communication [7]. As cache is usually small in size this is perfect for communication involving messages of small payload used in instant messaging. The example of In-Memory cache are Redis, Aerospike and SQLite.

2. OVERVIEW

The popular low latency Message Brokers include Amazon Kinesis, NATS Streaming, RabbitMQ [8], ZeroMQ, Redis [10], Apache Kafka [9], Microsoft event hubs and Google pub/sub. The above mentioned Message Brokers are used in social networks, user tracking, log analysis and aggregation, real time stream processing, enterprise wide applications, internet of things and autonomous vehicles. As evident these have wide range of applications and careful analysis of each system is required before determining the best Message Broker for our use case.

Below Redis, Apache Kafka and RabbitMQ are introduced to the reader giving the knowledge of the basics of these Message Brokers which helps in better understanding their key difference when we compare them in the next section.

2.1 Redis

Redis is an in-memory data structure store which can be also used as cache and message broker. We will look at the Message Broker part of redis which supports various data types including streams. Redis Message Broker implements a Publish/Subscribe paradigm of messaging complementing its in-memory cache features. Redis has channels that store the published data from publishers and consumed from using subscribers. Publishers use REdis Serialization Protocol (RESP) to publish data into one or more channels [10]. Order of data is preserved by Redis in a communication. Redis can be integrated with on-disk databases for persistence storage [10]. In-Built replication of messages is provided by Redis Message Broker using master-slave Fig.1, where data in master server is replicated and stored in the slaves. Redis implements pipelining to reduce the latency in messaging and decouple the subscribers and publishers. It can also batch the replies to save time. In-Memory cache access speed can be the differentiating factor from other Message Brokers in choosing it over others.

The dynamic network topology as well as the greater scalability offered by Redis is due to the decoupling of publishers and subscribers [10]. Channels can be subscribed and unsubscribed at any time, special messages are sent to indicate various operation by Redis which is detected by first element of the message. When channel receive these elements, they identify them as control operation and do as indicated.

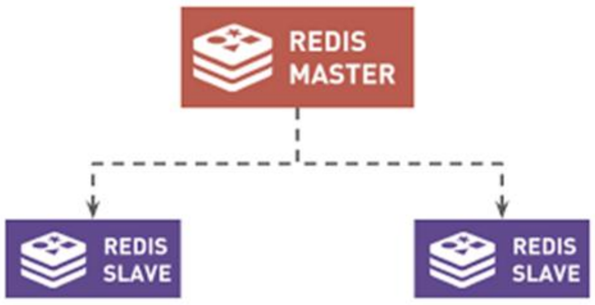

Fig -1: Redis Master Slave Architecture [21].

2.2 Apache Kafka

Apache Kafka is distributed message streaming platform [9]. It uses distributed Publish/Subscribe paradigm for messaging written in Scala language with latest releases it supports java as well. Kafka since inception has grown tremendously and can be seen used in companies like Uber, Spotify, Slack, Shopify, LinkedIn, Yahoo, Twitter and many others. Indicating the wide range of applications where Kafka can be used which are applications having real time processing constraints and low latency requirements. Important real life use cases of Kafka is as follows activity tracking, event sourcing, commit log, log monitoring, message replay, log aggregation, error recovery and the most important one which it identifies itself with in general platform for real time streaming. Kafka is simple to use as it is easy to setup and run.

Architecture of Kafka Fig. 2, few important terminologies to understand it's working is explained below:

- **Topic:** Kafka Topics is the place where data is pushed by producer and consumed by consumer. Producers and consumers can write and consume from multiple topics. Topics are further divided into partitions.
- **Producers:** Producers can publish data into chosen partitions of a topic [9].
- **Consumers:** Consumers have a group name which collectively consume from topics from same group name, they can be customized to consume from selected partitions [9].
- **Stream processor:** The messages pushed into Kafka topic is of form byte stream and stream processor helps in reading and writing messages to and from topic [9].
- **Broker:** Load balancing in Kafka is done by brokers as they can access multiple partitions simultaneously hence increasing the throughput [11].
- **Zookeeper:** Zookeeper is a service that suns on Hadoop stack which manages all the Kafka brokers [11].

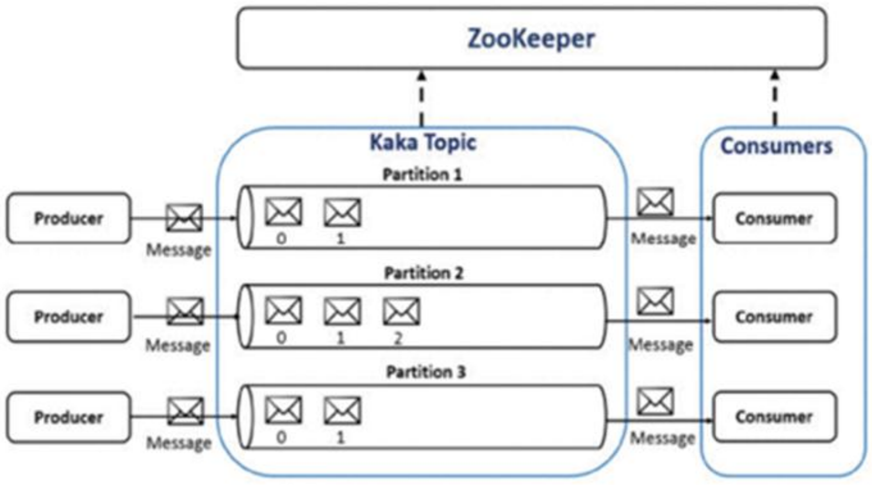

Fig -2: Apache Kafka Framework [11].

2.3 RabbitMQ

RabbitMQ is an open source Message Broker also one of the popular message broker system that is used for guaranteed message communication and queueing. Advanced Message Queuing Protocol (AMQP) is a standard protocol used by RabbitMQ for message brokering [8]. The data is transmitted using AMQP in two stages. First message is sent to an exchange. Secondly exchange forwards the messages to different available queues based on the selected exchange in first step. Copies of message can be broadcasted using exchanges which exist for that reason. Exchanges are capable of pattern matching addressing queues by name. On acknowledged by server in AMQP the messages are removed from the Queue. Queues can be persisted on disk so they are not lost on restart also they must be predefined before use. Replaying of messages is possible on slight modification of default configuration, replication can be enables by clustering of nodes.

New protocols such Streaming Text Oriented Messaging Protocol (STOMP) and MQ Telemetry Transport (MQTT) as have been added into RabbitMQ [8]. These protocols enable decoupling of the RabbitMQ broker hence the sender and receiver need not be online at the same time. RabbitMQ is language agnostic [8]. It runs on various Operating Systems with wide language supports and native cloud deployment.

TCP connections are used to send messaged. Sender, Consumers, Exchange and Queues are major components of RabbitMQ as shown in Fig. 3. Route of particular message are decided by binding. Payload and routing key are mandatory for published messages. The routing key is used to determine the queue where a particular message delivery has to be done.

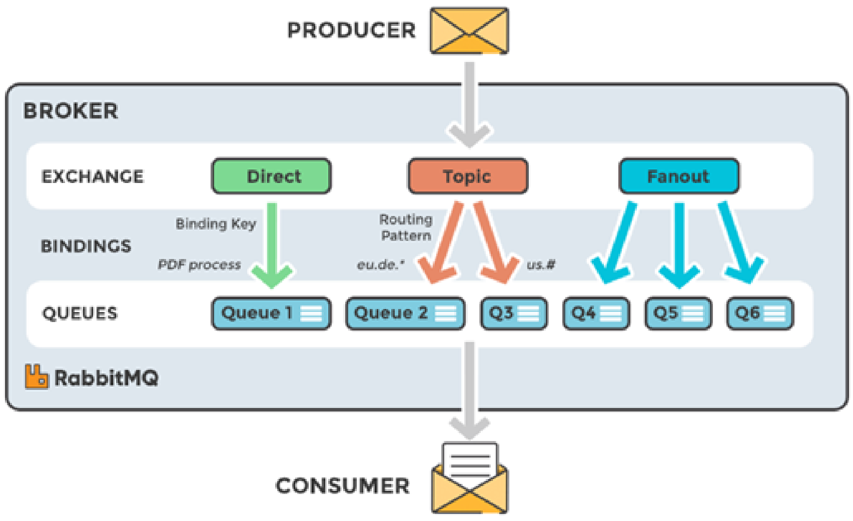

Fig -3: RabbitMQ Framework [22].

3. COMPARISON OF FEATURES

In this section, we are going to dive deeper into above described RabbitMQ, Apache Kafka and Redis to know their similarities and differences. Quality of Service (QOS) is generally used to denote the priority of messages as the messages are divided into various classes and these classes are assigned priorities which is crucial for latency time critical applications. The quality of service for above Message Brokers has already been done considering various factors such as throughput, latency, scalability in these papers [12], [13]. The QOS factors are considered after careful analysis of systems present in the industry and their requirement.

3.1 Message Delivery Guarantee

The guarantee that a particular message will be delivered is most crucial for applications that cannot lose a single bit of data and the data is very valuable. On the other hand video streaming, can afford to lose a byte or two. Thus, Message Delivery can be argued as most important consideration when choosing a Message Broker. Different type of message guarantees that are available for Message Brokers are as follows:

- At Most Once: The message is sent once if not delivered then no action is taken. Best for performance.
- At Least Once: Once the message is sent system verifies the acknowledgement if not successful repeats until it is in the process might create redundant messages.
- Exactly Once: The kind of message delivery we all expect message is delivered once with no redundant copies of it needs two way commit which is costly and bad for performance.

At least once delivery, at most once delivery and exactly once delivery can be opted in the case of Kafka hence, very flexible in terms of delivery as it can be customized according to the needs over other two message brokers [14]. At least once delivery and at most once delivery can be chosen in RabbitMQ [14]. At most once delivery which gives the highest throughput is the only option for Redis.

3.2 Message Persistence

If the system suddenly crashes and must be restarted or crashes permanently. This is when persistence of the message comes into picture. It denotes the capacity of Message Broker to recover from this type of crashes. More in detail:

- Failure of Temporary storage is when runtime memory is lost easy to recover.
- Failure of Permanent storage can only be recovered if backup or replica of the storage is kept

The messages in RabbitMQ can be stored either in a disk which is permanent storage that increases access time or in its memory which has a faster access time compared to disk [8]. For the message to persist through crash the user needs to explicitly configure in RabbitMQ [16]. The messages are stored on disk in terms of log that contains message and meta data about it also retention period can be set for message after which the message is automatically deleted, retention period has no limit as such in Kafka. Redis on the other hand does not provide any kind of message persistence [7]. Messages are lost on restart.

3.3 Message Ordering

Message ordering plays an important when we want to preserve the order in which the messages are sent and received. As it is very important in streaming or IOT systems that order is preserved. Few types of ordering are listed below:

- No Ordering: The messages are not ordered. High Throughput.
- Partitioned Ordering: Ordering is enforced only inside a partition. Throughput is affected slightly.
- Global Ordering: The messages are transmitted one after the other to get a Global Ordering. Throughput is bad in this case.

Partitioned ordering is maintained inside each channel of RabbitMQ. Concurrent write to multiple channels result in no ordering of messages in rabbitMQ. Similat to RabbitMQ the partitioned ordering is maintained inside the partition of a topic in Kafka. However, Global ordering can be performed in kafka by allowing only one publish request at a time which will adversely impact maximum performance [14] reducing throughput by significant amount. Redis maintains partitioned order when the message is published to channel sequentially and no ordering when messages are published concurrently to a channel.

3.4 Throughput

Throughput is the measure of how fast can data be transferred between producer and consumer with message broker as intermediate. Various experiments were carried out on Kafka, RabbitMQ and Redis which resulted in Redis being faster than Kafka and Kafka faster than RabbitMQ in the paper [15]. The throughput of kafka mainly depends on the configuration in which its running on, kafka takes a major hit in throughput for large message sizes with replication as suggested in paper [17]. RabbitMQ is in medium range in throughput even after disabling the acknowledgement for the messages due to deliverables it's just not fast as others [16]. Variations has been observed in performance of Kafka with change in number of broker and message size the ideal case scenario is considered above. Redis on the other hand has a very high throughput compared to other two as it is in memory cache and does not offer any kind of guarantee for message delivery.

3.5 Latency

The time between message sent by sender and received by receiver is called as latency usually called delay. Redis has a very low latency due to its architecture and model explained above [10]. Kafka provides low-latency service with wide variety of options for customizing the latency while balancing replication and others [9]. Compared to the other two RabbitMQ messaging queue is quite slow although newer protocols for at-most once delivery and at least once delivery in RabbitMQ [8]. If Kafka accesses messages from disk, then the latency would further rise as persistent access are slower. Redis has a very low latency as it is an in-memory cache message broker hence has smallest read and write time when compared to other two.

3.6 Availability

The actual time period in which the system is accessible out of given time period indicating the systems uptime. Highly Available (HA) systems are usually fault tolerant so we will see in this section how to make the system fault tolerant. The single queue in RabbitMQ gives the highest performance when compared with the mirrored queue which has replication factor of two decreased the performance and occupies twice the storage but gives better fault tolerance was seen in [18]. The zookeeper present in kafka is used to manage brokers, multiple brokers in kafka can be used in replicating the messages for better availability where even if a broker fails the data is not lost but we observe a performance dip as replication factor is increased also zookeeper coordinates between the brokers, producers and consumers and chooses another broker if one fails as observed in [19]. Kafka provides high performance along with replication. Redis follows master-slave replication, that means you can have one master and many. It allows distributing the load across machines and making use of memory of all the machines in the cluster (shared bandwidth) increasing availability. Kafka clearly offers wide range of Availability option among the three.

3.7 Scalability

It defines how well the system can cope with the changing environment such as increase in the number of messages produced and consumed handled by the Message Broker. Clustering can be used in RabbitMQ where multiple nodes acts as single broker which helps in balancing the workload on present on the system as well as scale the system when huge number of messages needs to be processed reliably [20].

Horizontal scaling was one of the fundamental ideas on which Kafka was hence, multiple brokers can be created to handle the increasing work load on each partition and performance can be adjusted with the load on the system brokers makes Kafka easy to scale. Redis can be scaled by sharding on the application side by using consistent hashing, this will allow to distribute your writes across multiple machines.

Table -1: Feature Comparison Table

Features	**Apache Kafka**	**RabbitMQ**	**Redis**
Developing Language	Scala	Erlang	C
Message Paradigm	Publish/Subscribe and Messaging Queue	Publish/Subscribe and Messaging Queue	Publish/Subscribe
Brokered/Brokerless	Brokered	Brokered	Brokered
Throughput	High	Varies between Medium and high	Very High
Latency	Low	Varies between low and medium	Very Low
Supporting Protocols	Binary protocol over TCP	MQTT, STOMP, AMQP	REdis Serialization Protocol (RESP, RESP3)
Supported Data Structure	Structured commit log	Queue	Strings, hashes, lists sets, bitmaps
Size of Message	1 MB max	2 GiB	512MB max
Message Delivery Guarantee	At Least Once, At Most Once, Exactly Once	At Least Once, At Most Once	At Most Once
Replication	Replicated cluster	Clustering	Master-Slave
Language Support	17 languages	30 languages	49 languages
Message Ordering	Yes	Yes	Yes
Persistence	Disk	In-memory (saving logs in files)	In-memory dataset and saving in disk
Basic Units	Topics	Queues	Channels
Company Footprint	Facebook, Netflix, LinkedIn, Twitter, Chase Bank	Reddit , Mozilla, 9GAG, MIT, AT & T,	Twitter, GitHub, Weibo, Pinterest, Snapchat, Stack Overflow, Flickr

4. DISCUSSION

The Various comparisons of features of Message Broker are briefly represented in Table I. As seen in the table there are few similarities between the Message Brokers which can further be studied to gain valuable insights. While designing cloud native solution, real time processing, big data, internet of things or microservices solutions hosted on cloud the table needs to be referred as it contains valuable insights which when ignored may cause system to fail or result in a poor design which should be avoided in software development at all times. Kafka as termed by creators is a streaming platform capable of handling large amount of data in real time or near real time. Kafka has seen the growth in recent years with big data, big data sources are popularly connected through Kafka which then at its own pace pushes data into Hadoop balancing the load splendidly. Kafka is also fluent with log related applications can be even distributed systems. Video streaming and surveillance, banking on a huge scale is easily handled by Kafka as small packets of data are pumped at very high throughput which is scalable as well. RabbitMQ is a Message Broker that comes as generic as possible which is one of the reason for its extreme popularity and wide range of applications. RabbitMQ has the capability to route the messages as required with the help of exchanges and queues which is much needed in the case of Remote Farming or Remote Driving on a pretty huge scale can also be used in internet of vehicles where direction is important. Huge Financial corporation rely on RabbitMQ for message delivery guarantees as it is RabbitMQ's forte to provide services in area where data is most important and should work without any loss in data. As Redis is and In-Memory cache it can be used in embedded industry for application that process small amount of data but at a very rapid pace like graphics card, Arm board. Redis is also very good at handling images and document hence various images, videos and documents can be uploaded in real time and viewed as well. Posts and Comments in facebook, instagram, stack overflow that appears within blink of an eye is all thanks to the Redis working in the background especially popularly used in social media application. Redis is also preferable in use cases where messages can be sacrificed for the sake of high throughput like video streaming.

5. CONCLUSIONS AND FUTURE SCOPE

This paper starts by giving a concise introduction on Publish/Subscribe Paradigm, Message Queue and In-Memory Cache to the readers helping them know the design options that are available to them in the world of Message Brokers. Out of many Message Brokers present out there three of the most widely used and popular Message Brokers Redis, RabbitMQ and Apache Kafka are selected to be dwelled into deeper which represent their messaging models. The introduction on Redis is given and then it is explained how it can be used as a Message Broker that follows Publish/Subscribe Paradigm keeping the benefits of In-Memory Cache and its applications. Kafka is also introduced to the user with a slight peak at its architecture, basic terminologies and applications where it can be used to enhance the solutions in various domains. Finally, RabbitMQ is introduced to the user which offers general message queueing as well as advanced message routing techniques giving reader the complete feel on the options that are available from which readers can select the Message Broker that suits their solution. Considering the requirement and deliverables in the industry, features are selected which would affect the choice of selection of Message Brokers the most. Deeper insight is given on these features with respect to the Message Brokers selected and choice to make when one of them is single most important requirement to the reader. The quick glance at the Table provides reader with clarity on what Message Broker fits their needs. Hence, readers are to refer this paper when they want a quick introduction into the world of Message Brokers and to look at the variety it has to offer to the various domains of Manufacturing, E-Commerce, Finance, IOT and Big Data. The paper can also be read by knowledgeable readers when they are confused as to which Message Broker fits well for their use case or which Message Broker they must choose with luxury of choice.

It is not evident at start but Message Brokers hold the key to the Low Latency communication which is much needed in this growing world of big data, it forms the core of big data streaming and processing, real-time analytics, IOT and microservices domain as more and more data is generated and applications are built to handle and analyze the data to gain meaningful insights. Future work on the paper would be to further dive deeper into the individual message paradigm and track their changes with every new release. Also, studying any new Message Broker that arrive in the world of Message Brokers to compare those to the drivers of the industry to give the readers clarity on what Message Broker best suits their use case.

ACKNOWLEDGEMENT

This work would not have been possible without the support by Dept. of Computer Science and Engineering Department, R V College of Engineering, Bengaluru.

REFERENCES

1. Z. Peng, "Stocks Analysis and Prediction Using Big Data Analytics," 2019 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS), Changsha, China, 2019, pp. 309-312.
2. R. Rocha, L. L. Ferreira, C. Maia, P. Souto and P. Varga, "Improving the performance of a Publish-Subscribe message broker," 2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC), Valencia, Spain, 2019, pp. 91-92.
3. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. “The Many Faces of Publish/Subscribe” ACM Computing Surveys (CSUR), vol. 35, pp. 114–131, 2003.
4. Gracioli, Giovani & Dunne, Murray & Fischmeister, Sebastian “A Comparison of Data Streaming Frameworks for Anomaly Detection in Embedded Systems” 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec),2018.
5. S. Stoja, S. Vukmirovic and B. Jelacic, "Publisher/Subscriber Implementation in Cloud Environment," 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Compiegne, 2013, pp. 677-682.
6. V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), Craiova, 2015, pp. 132-137.
7. P. Zhang, L. Xing, N. Yang, G. Tan, Q. Liu and C. Zhang, "Redis++: A High Performance In-Memory Database Based on Segmented Memory Management and Two-Level Hash Index," 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social

Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), Melbourne, Australia, 2018, pp. 840-847.

8. RabbitMQ website, April 2020. Available online: https://www.rabbitmq.com/.
9. Apache Kafka website, April 2020. Available online: https://kafka.apache.org/.
10. Redis website, April 2020. Available online: https://redis.io/.
11. B. R. Hiraman, C. Viresh M. and K. Abhijeet C., "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.
12. P. Bellavista, A. Corradi and A. Reale, "Quality of Service in Wide Scale Publish—Subscribe Systems," in IEEE Communications Surveys & Tutorials, vol. 16, no. 3, pp. 1591-1616, 2014.
13. F. Araujo and L. Rodrigues, "On QoS-aware publish- subscribe," Proceedings 22nd International Conference on Distributed Computing Systems Workshops, Vienna, Austria, 2002, pp. 511-515.
14. P. Dobbelaere and K. S. Esmaili. "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations" In Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, pages 227–238. ACM, 2017.
15. T. Treat. Benchmarking Message Queue Latency, http://bravenewgeek.com/benchmarking-message-queue-latency/, 2016
16. 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
17. 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.
18. 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
19. 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.
20. Martin Toshev, "Learning RabbitMQ" Birmingham, UK: Packt Publishing, Ltd, 2015
21. Opstree blog, April 2020. Available online: https://blog.opstree.com/2019/10/22/redis-cluster-architecture-replication-sharding-and-failover/.
22. CloundAMQP blog, April 2020, Available online: https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html.