IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 2, 2020, accepted December 15, 2020, date of publication December 22, 2020,
date of current version January 4, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3046503

A Fair Comparison of Message Queuing Systems

GUO FU, YANFENG ZHANG, (Member, IEEE), AND GE YU, (Senior Member, IEEE)
Department of Computer Science and Engineering, Northeastern University, Shenyang 110819, China

Corresponding author: Yanfeng Zhang (zhangyf@mail.neu.edu.cn)

This work was supported in part by National Key R&D Program of China (under Grant 2018YFB1003404), National Natural Science Foundation of China (under Grants 62072082 and 61672141) and Fundamental Research Funds for the Central Universities (under Grant N181605017, N181604015, and N181604016), and Key R&D Program of Liaoning Province 2020JH2/10100037.

ABSTRACT The production and real-time usage of streaming data bring new challenges for data systems due to huge volume of streaming data and quick response request of applications. Message queuing systems that offer high throughput and low latency play an important role in today's big streaming data processing. There are several popular message queuing systems in production usage and also many in-lab message queuing systems in academia. These systems with different design philosophies have different characteristics. It is non-trivial for a non-expert to choose a suitable system to meet his specific requirement. With this premise, our primary contribution is to provide the community with a fair comparison among message queuing systems, using a standardized comparison metric and reproducible experimental environment. Five typical message queuing systems (including Kafka, RabbitMQ, RocketMQ, ActiveMQ and Pulsar) are evaluated qualitatively (in analysis) and quantitatively (in experimental results). This article also highlights the distinct features of each system and summarizes the best-suited use cases of each system. The fair comparison and the insight analysis provided in this article can help users choose the best-suited message queuing systems.

INDEX TERMS Big data, streaming processing, message queuing system.

I. INTRODUCTION

In the era of information explosion, huge amounts of data are being produced, transmitted and consumed continuously every day. Streaming data are generated continuously by thousands of data sources, which typically send data records simultaneously. Streaming data include a wide variety of data such as log files, online purchase records, geospatial data, information from social networks, and financial trading floors. The production and real-time usage of these streaming data bring new challenges for data systems due to its huge volume and quick response time request. Traditional distributed file systems (e.g., HDFS [1]), cloud storage systems (e.g., Amazon S3 [2]), and key-value store systems (e.g., Apache Cassandra [3]) are not competent to support real-time processing of these streaming data [4]. The distributed message queuing systems play an increasingly important role in streaming data processing applications, such as high quality real-time search, analysis, and recommendation services.

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

Message queuing systems store messages (data) and allow different terminals to connect to the queue to publish or consume messages. The distributed message system consists of multiple *Brokers* (worker machines) that work together to provide the message service. The entity that publishes messages to the system is named *producer*, and the entity that subscribes messages from the system is named *consumer*. The messages are grouped into *Queues* or *Topics* to separate the messages that belong to different producers/consumers. Messages sent by the producer must specify which Topic they are sent to.

Message queuing systems can improve stream processing in the following four scenarios: (1) **Asynchronous Processing.** In synchronous processing, the data producer needs to receive a response from the data consumer before sending the next data record. With a message queuing system, the producer sends a message (i.e., data record) to the system and then continues to produce next message without synchronizing with the consumer side, which avoids wasting resources while waiting for the feedback. (2) **Decoupling of Producers and Consumers.** In practice, a system communicates with multiple systems through different interfaces. The coupling of producer-consumer systems increases the

burden of programmers since they have to handle the connections to different systems. With a message queuing system as middleware, the producer is only responsible for publishing the message to the queue without any consideration of the diverse communication mechanisms with different consumers. (3) **Peak Shaving.** In the high concurrency scenario, servers can crash because of too many requests in a short time. A message queuing system can be introduced to buffer the data, where consumers can pull the message data according to their own processing capacities. (4) **Log Service.** The logs of some production systems are potentially extremely large. Message queuing system can be used to provide log service, responsible for receiving, storing and forwarding log data and offering guarantees on the strict order based on log generation [5].

Many message queuing systems have been proposed and widely used, such as Kafka [6], RabbitMQ [7], RocketMQ [8], ActiveMQ [9], and Pulsar [10], etc. These systems have different design architectures and have their own characteristics, which makes them suitable for different scenarios. However, it is non-trivial for a non-expert to choose a suitable system according to his specific requirement. Although there are prior works on system comparisons, we think they have some shortcomings: (1) Only 2-3 system candidates are selected for comparison, and the comparison is not comprehensive enough. (2) The functions of system might be overestimated from its official report, and there is no fair experimental comparison to quantitatively compare performance differences.

Specifically, this article makes the following contributions: First, We summarize the main features of a message queuing system as our evaluation metrics (in Section 2), including production features, quality-of-service guarantee features and performance related features. Second, We introduce and analyze five popular message queuing systems (in Section 3), including Kafka, RabbitMQ, RocketMQ, ActiveMQ and Pulsar. We choose them because 1) these systems are relative popular, and their communities are more active than others. 2) these message queuing systems are representative in multiple aspects, such as implementation language (Java, Scala, or Erlang), functionality support (batching, priority queuing, and delay queuing), and communication protocol (TCP, AMQP, Customized, and Hybrid). According to the features we summarize in Section 2, we analyze the architecture of each message system, and as a summary and supplement to the differences of these systems, we offer a table at the end of this section. Third, we develop a test framework to fairly evaluate the performance of the system, we also design several experiments to compare the performance of Kafka, RabbitMQ,RocketMQ,ActiveMQ,Pulasr with various parameters(in Section 4).

The rest of the article is organized as follows. Section III summarizes the main features of message queuing systems. Section IV analyzes the pros and cons of Kafka, RabbitMQ, RocketMQ, ActiveMQ, and Pulsar. Section V measures the performance features of these message queuing systems. Section VI offers several guidelines to choose the best-suited system given different application requirements. Section II presents some work related to the article. Section VII concludes the article.

II. RELATED WORK

In this section, we present existing studies that discussed the message queuing systems. The authors of [11], [12] compared message queuing systems qualitatively. [11] focused on system usage comparison, including installation and documentation, etc. The communication and architecture of the several message queuing systems are compared in [12]. The performance comparison between RabbitMQ and ActiveMQ is studied in [13]. The performance comparison between Kafka and RabbitMQ is studied in [14] and [15]. A quantitative and qualitative evaluation is performed in [5], [16], but their experimental results are from the built-in test tools of each system, which may lead to fairness issues.

Most of the existing studies are focused on only two or three message queuing systems. To the best of our knowledge, there is no detailed study that provides a multi-dimension comprehensive comparison on the five popular systems. In addition, though the quantitative comparison is performed for the test systems, their experimental results are obtained by using the built-in test tools that provided by the systems. This is not fair and convincing enough. We fill this gap by developing a test framework and performing a fair comparison. We also discuss the best-suited use cases for different systems, which will help users choose a message queuing system according to the application requirement.

III. MESSAGE QUEUING SYSTEM FEATURES

In this section we summarize the main features of message queuing systems, which can be used to establish a common framework for comparison between message queuing systems.

A. PRODUCTION FEATURES

We summarize a few production features and design choices of the message queuing systems as follows:

1) DEVELOPMENT LANGUAGE

Different message queuing systems use different development languages, the characteristics of language will also bring corresponding advantages to systems. For example, due to the widespread use of the Java language, the message systems implemented by Java are also very convenient for developers to conduct secondary development, and as a result they usually have faster updates.

2) COMMUNITY ACTIVITY

Community activity level implies the popularity of a message queuing system. Popular products will have a better integration and compatibility with the surrounding ecosystem. The more active community can have more contributors

discovering bugs and fixing bugs, and as a result the system continues to be more stable and be faster improved.

3) COMMUNICATION PROTOCOL CHOICE

Message protocol needs to be applied when realizing message queuing function. The protocols used in each message system are different. According to whether the message protocols used are open to industry, they can be classified into open protocols and private protocols. The widely used open message protocols include AMQP [17], XMPP [18], REST [19], and STOMP [20]. Some systems extend basic protocols according to their own requirements, while some use customized messaging protocols. These protocols are called private protocols.

4) CONSUMPTION MODE

There are two modes of consumption: Push and Pull. (1) Push mode requires that the message system actively pushes messages to the consumer. This mode has better real-time performance, but requires a certain flow control mechanism. Once the system pushes a large number of messages to the consumer, it will incur heavy load to consumer to make it slow or even crash the service. (2) Pull mode refers to that the consumer actively requests pull messages from the message system. In this mode, the consumer can consume message data according to its own capacity and send pull request to the system every other time interval. However, this requires consumer to set a reasonable message pull interval, which is not trivial. Too frequent pull requests will incur significant heavy burden to the message system, but if the pull requests are not frequent enough it would inevitably cause data latency problem.

5) SYSTEM ARCHITECTURE

Distributed message queuing systems typically have two architectures: master-slave and peer-to-peer. In master-slave architecture, Brokers are divided into a master and multiple slaves. The master provides service and the slaves serve as its backup. Once the master goes down, the slaves can take over related service. In peer-to-peer architecture, all Brokers have the same status, and each message is backed up in multiple Brokers to be resistant to message loss and machine failure.

6) MESSAGE QUEUING MODEL

There are two message queuing models in message service: point-to-point and publish-subscribe. In point-to-point model, every message will be sent to a specific Queue, only one corresponding consumer can obtain this message. In publish-subscribe model, every message has a category called Topic, a subscriber to a Topic can consume all of its messages, and a Topic can be subscribed by multiple consumers.

7) USABILITY

The usability of a system is also an important production feature, which helps them get start easily. The usability can be evaluated in several aspects, such as the easiness of installation, the completeness of documentation, the functionalities of management and monitoring, etc. A system with high usability can attract more users by saving their time on learning to use and maintain it.

8) COMPATIBILITY

Message queuing system can play the role of message middleware for many-task computing on a big-data platform [21]. Therefore, compatibility with other tools such as storage and processing is also an important feature that should be considered. We mainly analyze the compatibility of message queuing systems with the mainstream storage system HDFS [1] and the big data processing system Spark [22] and Flink [23].

B. QUALITY-OF-SERVICE GUARANTEE FEATURES

As a middleware system providing message buffering service, the message systems are defined by several required and desired guarantees. Especially for distributed message systems, these quality-of-service guarantees become more important in production usage.

1) DELIVERY GUARANTEE

To ensure that messages are transmitted between producers and consumers, there are generally three delivery guarantees: (1) at-most-once: Messages are transmitted at most once, they may be lost, but will never be transmitted repeatedly. (2) at-least-once: Messages are transmitted at least once, they will never be lost, but may be repeated. (3) exactly-once: Every message must be transmitted once and only once. For most message queuing systems, only at-most-once guarantee or at-least-once guarantee is provided [24]. It is difficult to realize exactly-once guarantee.

2) ORDERING GUARANTEE

In a message system, it is very important to ensure that messages are sent and consumed in the same order. In some application scenarios, the order of messages is very strict. There are three ordering guarantee variants: (1) no-ordering: This is an ideal case for performance. Because no ordering guarantee is provided, no additional resources are used in this mode. (2) partition(or queue)-ordering: In some message queuing systems, a partition is the basic unit for parallel operations. In this mode, messages are guaranteed to be ordered within a partition, but it is not guaranteed across partitions. Though requiring more resources than the previous mode, it can also have high performance. (3) global-ordering: In order to provide a global ordering guarantee across different channels, it requires lots of resources to keep synchronization between producers and consumers, which significantly degrades the performance of message systems.

3) RELIABILITY

Message queuing systems should be strong against software and hardware failures. If one or more machines are down,

other machines can be used without affecting the system. This relies on the synchronization and replication of messages, which will take up additional hardware and software resources.

4) SCALABILITY

The scalability means that a message queuing system can continue to scale as demand increases. For instance, the processing ability can be easily changed by adjusting the number of messages, partitions, or producers/consumers. This can be achieved by adding new machines (or other hardware resources) to the system.

5) TRANSACTION SUPPORT

The transaction requirement requires that: either a complete sequence of messages is sent (received), or none of them is. For instance, the transaction can be successfully committed only after the consumer has consumed the entire message queue.

C. PERFORMANCE FEATURES

Regarding the performance features, two key metrics are the latency (or response time) and the throughput [15].

1) LATENCY

Latency measures how long it takes for a message to be transmitted between endpoints. In message systems, typically the main latency contributors are as follows [16]: (1) The calculation cycle required for packet metadata processing, such as validating, routing, which is usually independent of the packet size. (2) The calculation cycle required for packet replication, which is usually related to packet size. (3) Memory access latency. The level of latency depends on the access operation and its location and method. For example, write versus read, DRAM versus disk, and sequential access versus random access. (4) Overhead for some specified cases. In order to ensure the reliability of message transmission, the message queuing provides some special guarantee mechanisms as mentioned in previous subsection, which will bring extra overhead and increase the latency. For example, the three ordering guarantee modes have varying degrees of impact on performance, thus affect latency. (5) Dequeuing latency. When a queue is not empty, it has a behavior of FIFO (fist-in-first-out), which will bring the dequeuing latency.

Low latency describes a message queuing that is optimized to process a very high volume of data messages with minimal delay (latency). These systems are designed to support operations that require near real-time access to rapidly changing data and bring a good user experience.

2) THROUGHPUT

Throughput means the number of messages or message bytes that can be pumped through a message queuing system per time unit. High throughput means that messages are less likely to be backlogged. The five factors that affect latency discussed above also affect throughput. Experienced system designers use various techniques to improve throughput, such as batch processing. But it is known that there is a fundamental trade-off between latency and throughput with stream processing style optimizing for latency and batch processing style optimizing for throughput. However the common question to ask is how much data to batch together to get good throughput? Because this trade-off is important, most message systems such as Kafka [25] allow users to specify how much data they want accumulated on the system side before the system should complete user's fetch request. In Section V, we will evaluate the throughput and latency of each system by varying the batch size.

IV. REVIEW OF MESSAGE QUEUING SYSTEMS

In this section, we give brief description of five typical distributed message queuing systems including Kafka, RabbitMQ, RocketMQ, ActiveMQ, and Pulsar. We also compare them in different aspects and provide a summary table to highlight the main difference.

A. KAFKA

Kafka [6] is a distributed, separable, redundant, and persistent logging service developed by Scala based on the TCP protocol. Kafka has an active community, which makes it more and more popular these days. In Kafka, messages are classified into different Topics, and each Topic can be divided into multiple partitions, which are distributed and stored on different Brokers (a cluster of worker machine). By introducing the partition concept, the messages are uniformly distributed to multiple partitions, which help improve the parallelism of the system [26]. In addition, multiple consumers can simultaneously read messages from one or more partitions to improve parallel processing capabilities.

As shown in FIGURE 1, there are multiple producers, brokers, consumers and a Zookeeper [27]. Kafka employs a peer-to-peer architecture rather than a master-slave architecture, where all brokers maintain the same status. Producers use the Push mode to publish messages to the Broker, and all consumers in a group work together to consume all partitions of a subscribed Topic by Pull mode. Each partition is a sequential,

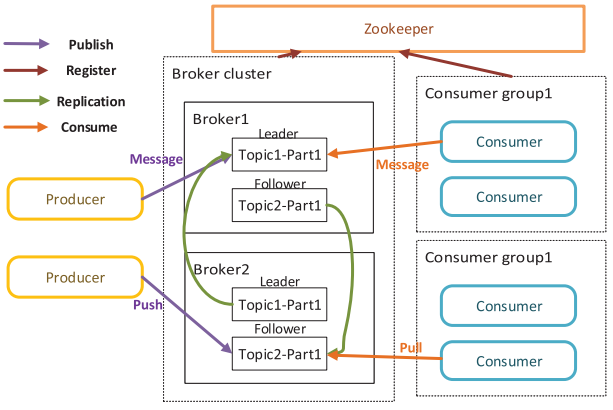


FIGURE 1. Kafka architecture.

immutable message queue that can be added continuously. Messages in a partition are assigned a sequence number, called offset, which is unique in each partition. Under normal circumstances, the offset increases linearly as the consumer consumes the message. But the actual offset is controlled by the consumer, who can reset the offset to re-read the message. Each partition can only be digested by a single consumer in a consumer group and the messages in a partition are guaranteed to be ordered. The producers are not necessary to be managed by the cluster resource manager Zookeeper. This is because that producers are transient and can be closed at any time without coordination. But the Consumers and the Brokers need to be managed by Zookeeper to achieve load balancing.

The high availability/reliability of Kafka is achieved through backup mechanism (i.e., each partition has several replications), and the replicas are stored on different Brokers and are synchronized. There is a leader replica and multiple follower replica, and all the write/read loads are directed to the leader replica. Once the worker where the leader replica locates fails, a new leader will be elected from the followers to continue the service. In addition, Kafka supports transactional message and delivery guarantee. It support at-least-once delivery guarantee by default, and allows at-most-once guarantee by setting the producer to submit asynchronously. Exactly-once guarantee can also be implemented but requires cooperation with the target storage system.

When using earlier versions of Kafka, users need to install Zookeeper first. In the latest version, Kafka has built-in Zookeeper, which eliminates the installation steps. Kafka has a very detailed document. Users can easily and quickly learn how to use different versions of Kafka. However, Kafka relies on logs for management and monitoring, so it does not have a very user-friendly interface. In addition to persisting messages to log files, Kafka is also compatible with HDFS, it provides a connector called *Confluent HDFS Connector* for writing data from Kafka to Hadoop HDFS. Kafka is easily integrated into stream processing systems. Flink can consume messages from Kafka by *flink-connector-kafka*. Similarly, Spark can use Kafka as a data source through a tool called *spark-streaming-kafka*.

1) PROS AND CONS

Kafka is famous for its high throughput, which can be attributed to the following three factors. 1) Kafka makes use of zero-copy to avoid repeated copy operations [6]. The traditional data read from disk and data sending to the network involves 4 copy operations: a) from disk to page cache, b) from page cache to user space, c) from user space to socket buffer, and d) from socket buffer to network. The zero-copy technology only copies the data of the disk file to the page cache once, and then sends the data from the page cache directly to the network, avoiding repeated copy operations to improve performance. 2) Kafka uses batching of data to improve the throughput and reduce the RPC overhead [6]. 3) Kafka employs data compression techniques to improve the efficiency of data transmission [28]. It supports three compression algorithms: GZIP, Snappy, and LZ4.

B. RabbitMQ

RabbitMQ [7] is an open source implementation of AMQP protocol developed with Erlang(a multipurpose programming language for developing concurrent and distributed systems) and also has a very active community. Erlang inherently supports distributed computation (by synchronizing the cookies in the Erlang cluster nodes), so RabbitMQ does not rely a third-party cluster manager like Zookeeper. RabbitMQ cluster has two modes: normal cluster mode and mirror cluster mode.

As shown in FIGURE 2, RabbitMQ achieves high availability through Queue Mirroring. The Queue concept in RabbitMQ is similar to Kafka's partition. Each mirrored queue consists of one master and one or more mirrors where each Broker contains all the data in a Queue. The master is hosted on one node commonly referred as the master node. Each Queue has its own master node. All operations for a given queue are first applied on the Queue's master node and then propagated to mirrors. Once failure occurs, consumers can be forwarded to digest data from the mirror queue in other Brokers by Push mode or Pull mode. Exchange can be used as a forwarding agent, which helps implement route conversion and forward messages to the corresponding queue. After Exchange is bound to a queue, messages are distributed to the message queue according to different binding rules according to the type of Exchange, either one message is distributed to multiple message queues or one message is distributed to a message queue.

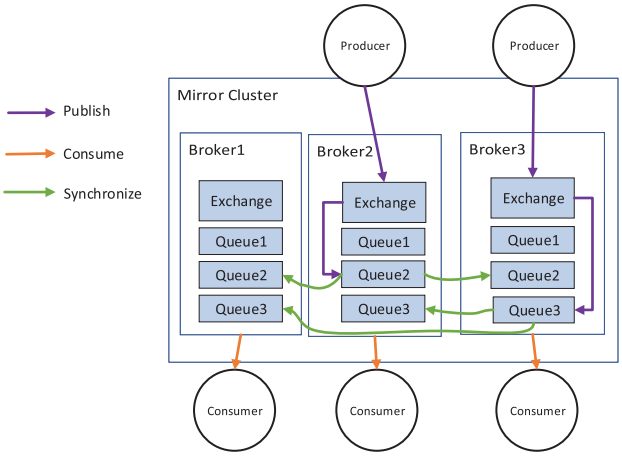


FIGURE 2. RabbitMQ architecture.

In a RabbitMQ cluster, there can be two type of nodes, memory node and disk node. There is at least one disk node in the cluster, on which configuration information and meta-information are stored. It should be noted that RabbitMQ's architecture has poor scalability due to the complete replication design.

RabbitMQ provides a Web UI for the system. Users can easily manage the queue and monitor the status of the cluster. It is also easy to install. The documentation tutorials cover the basics of creating messaging applications. RabbitMQ does not support persisting messages to other storage systems, such as HDFS. In previous versions of Spark, the *MQTTutils* tool can directly create rabbitmq data sources. However, Spark removed this tool after version 1.6. Flink and RabbitMQ are compatible via the connector *Flink-Connector-RabbitMQ*.

1) PROS AND CONS

RabbitMQ is a general-purpose message Broker which supports various standardized protocols and support multiple functionalities, such as priority queuing, delay queuing. Many useful plugins are provided which makes it convenient to manage. However, it is not as fast as Kafka because of its transactional mechanism. Publisher Confirm mechanism is proposed to solve the problem of transaction overhead and at the same time guarantee message delivery(it supports at-least-once and at-most-once). RabbitMQ can persist messages to disk, but the ordering of messages is not guaranteed.

C. RocketMQ

RocketMQ [8] is developed in Java, it is first used in Alibaba and later opensourced to public. RocketMQ is also supported in Aliyun and supports a set of customized communication protocols.

The architecture of RocketMQ is shown in the FIGURE 3. A set of NameServers, working as the cluster manager, compose a lightweight Topic routing registration center that supports dynamic Broker registration and routing query (from producers and consumers). RocketMQ has the concept of queue in each Topic, which is similar to the partition concept in Kafka. Multiple queues may exist on a Broker. The messages in the queue are guaranteed to be ordered. When sending messages, the user only specifies Topic, and the Broker selects which queue to be sent to according to the routing information of Topic. When a consumer subscribes to messages, it decides which queue messages to subscribe to based on the load balancing policy.

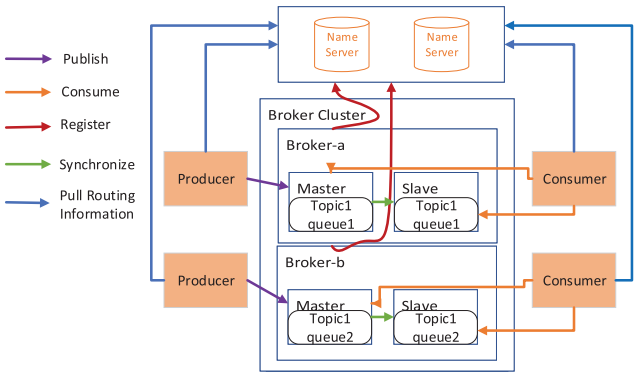


FIGURE 3. RocketMQ architecture.

To ensure reliability, RocketMQ uses a master-slave architecture. A cluster of Brokers consist of multiple groups of master/slaves. Master can read and write, but slaves can only read, where master synchronizes the written data to the slaves. If a master fails, consumers can still consume messages from slaves (by Push or Pull mode), but producers cannot write messages to it. All the data are persistence on disk, and messages are guaranteed to be delivered at least once. RocketMQ supports delay queuing and implements batch transmission, as well as transactional message. In addition, RocketMQ is easy to scale.

When using RocketMQ, you need to start the *namesrv* service before starting the broker, and you can only monitor the system status through the log output of the current directory. Some contributors in the RocketMQ community have developed a simple management interface that users can install by themselves. In the documentation, a user guide is provided, including examples of using various functions, such as batch processing, message filtering. There are many RocketMQ-related projects contributed and maintained by the community. These projects include many tools to integrate with other stream processing systems, such as Flink (*RocketMQ-Flink*) and Spark (*RocketMQ-Spark*). Although RocketMQ does not support HDFS, it is compatible with the mainstream database systems.

1) PROS AND CONS

RocketMQ stores all messages in the same physical file. The centralized store design improves the efficiency when increasing number of topics. As will be shown in our experimental evaluation, the increasing number of topics/partitions will cause a drastic decline of Kafka's throughput, while Apache RocketMQ delivers a stable performance. Therefore, Kafka is more suitable for business scenarios with only a few topics and consumers [29], while Apache RocketMQ is a better choice for business scenarios with a large number of topics and consumers [30]. However, RocketMQ community is not very active comparing to Kafka and RabbitMQ.

D. ActiveMQ

ActiveMQ [9] is written in Java together with a full Java Message Service (JMS) client supporting a variety of communication protocols, such as OpenWire, STOMP, REST, XMPP, and AMQP.

FIGURE 4 shows the ActiveMQ cluster architecture. Zookeeper is used to manage the registered Brokers. Only one Broker (i.e., master broker) in the cluster can provide service, and the other Brokers are slave brokers. Slave brokers synchronize with the master Broker. Once the master cannot serve due to a failure, Zookeeper will elect a new master from slaves to provide service. Different from other message systems, ActiveMQ supports both publish-subscribe mode and point-to-point mode. The Topic concept (similar to Kafka) in ActiveMQ corresponds to the publish-subscribe mode, while the Queue concept (similar to RabbitMQ) corresponds to the point-to-point mode.

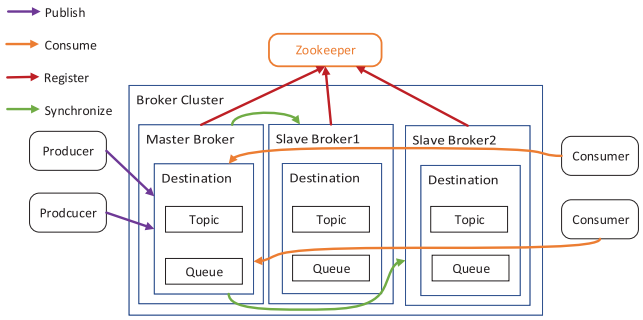


FIGURE 4. ActiveMQ architecture.

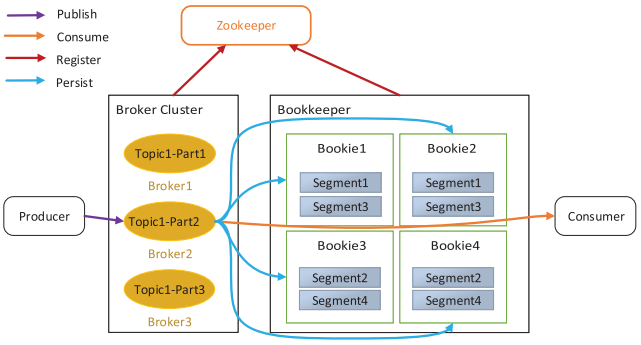


FIGURE 5. Pulsar architecture.

ActiveMQ manages Queue and Topic differently. The messages ordering in a Queue can be guaranteed by setting Exclusive Queue, while there is no ordering guarantee in Topic. There is at-least-once delivery guarantee for Queue, while there is no delivery guarantee for Topic. Messages in a Queue will be stored on disk file or in database, while messages in Topic are not persistent by default.

It is easy to install ActiveMQ, but its documentation is not detailed enough. There is no running samples to help users learn how to use it. ActiveMQ provides a simple interface to monitor and manage the system. ActiveMQ provides a variety of built-in storage engines, but it cannot persist data to storage systems such as HDFS. ActiveMQ cannot be used as a data source for Spark. Flink is compatible with ActiveMQ 5.14.0 by the connector *flink-connector-activemq-2.11*.

1) PROS AND CONS

ActiveMQ provides priority queuing, batch transmission, etc. It supports transaction message and scales well. However, its community activity is declining, there are less and less maintenance records and user feedbacks. RabbitMQ is lack of sharding function, this is a missing feature, due to JMS does not specify the sharding mechanism of messaging middleware, users must implement it their own as needed.

E. PULSAR

Pulsar [10] is an open-source large-scale distributed pub-sub messaging system originally created at Yahoo and now part of the Apache Software Foundation, it is developed by Java based on the TCP protocol.

As FIGURE 5 shows, Apache Pulsar's design architecture is fundamentally different from that of other messaging solutions, including Apache Kafka. Pulsar was designed with a layered and fragmented architecture to provide better scalability, and flexibility. Pulsar separates message storage from message service and provides good scalability. Broker receives messages from the producer and pushes it to the consumer, but the message is stored in Bookies, which are managed by Bookkeeper [31]. Pulsar stores Topic in logical units of partition like Kafka. The difference is that Pulsar's partition takes Segment as the physical storage unit. Each partition will be divided into multiple Segments, and a Segment will be stored in different Bookies. Since there is no Topic data stored in Broker, once a Broker is down, other Brokers can quickly take over its work. If a Bookie is down, another Bookie will read data from multiple Bookies and recover data on the failed node.

Pulsar has multiple QoS guarantees, including global ordering guarantee and delivery guarantee, it supports all the three delivery guarantees we introduced before (the exactly-once is called effectively-once in Pulsar). In addition, it supports priority queuing and batch transmission. As Pulsar is a new generation of message system, the community is growing and needs to be further tested by the market.

Pulsar provides s very complete and detailed documentation, covering multiple aspects, such as functions, concepts, architecture, and deployment. However, it does not provide a user interface to manage and monitor the system. In addition, the installation of Pulsar is more complicated because it relies on Zookeeper and Bookkeeper. The *Hdfs Sink Connector* is used to pull messages from Pulsar topics and persist the messages to a HDFS file. The *Spark Streaming receiver* for Pulsar is a custom receiver that enables Spark to receive data from Pulsar. Currently, Flink cannot use Pulsar as its data source. There is no official connector or library available.

1) PROS AND CONS

The most obvious benefit of the Brokers-Bookies separated architecture is the scalability. It also provides many useful function, include delay queuing, batching and priority queuing. Pulsar takes advantage of the multi-core advantage of modern systems to assign the same task request to the same thread, avoiding the overhead of switching between threads as much as possible. However, instead of searching messages locally, Broker needs to connect to another Bookkeeper cluster, which increases network overhead and has an impact on performance.

F. SUMMARY

TABLE 1 summarizes the main features of the 5 message systems. The first three rows depict the number of starts, the number of forks, and the the number of reported issues retrieved from GitHub, which imply the popularity of these systems. We can see that Kafka has attracted much more attentions than other systems. We classified the other features into three categories, production features (i.e., 4-12 rows),

TABLE 1. Message Queuing Systems Comparison

System	Kafka	RabbitMQ	RocketMQ	ActiveMQ	Pulsar
Stars	17.3K	7.8K	12.8K	1.8K	6.8K
Forks	9.3K	2.9K	7K	1.2K	1.7K
Issues	764	193	169	55	793
Develop language	Scala	Erlang	Java	Java	Java
Comm. protocol	TCP	AMQP	Customized	Multiple	TCP
Cluster manager	Zookeeper	Erlang	NameServer	Zookeeper	Zookeeper
Architecture	P2P	master-slave	master-slave	master-slave	P2P
Queuing model	Pub-Sub	P2P	Pub-Sub	Pub-Sub&P2P	Pub-Sub
Consume Mode	Pull	Push/Pull	Push/Pull	Pull	Push
Persistence	Disk	Mem/Disk	Mem/Disk/DB	Mem/Disk/DB	Mem/Disk
Usability	Medium	High	High	Medium	Medium
Compatibility	Excellent	Good	Excellent	Poor	Good
Deliver guarantee	ALL	at-least/most-once	at-least-once	at-least-once	ALL
Order guarantee	Partition-order	No-order	Queue-order	Queue-order	Global-order
Reliability	High	High	High	High	High
Scalability	Good	Poor	Good	Good	Excellent
Transactions	Yes	Yes	Yes	Yes	No
Batching	Yes	Yes	Yes	Yes	Yes
Delay queuing	No	Yes	Yes	No	Yes.
Priority queuing	No	Yes	No	Yes	Yes

quality-of-service guarantee features (i.e., 13-17 rows), and the additional functionalities supported by these systems (i.e., 18-20 rows). In addition to the above message queuing systems, there are other systems. ZeroMQ [32] acts as a concurrency framework and carries atomic messages across various transports like in-process, inter-process, TCP, and multicast. OpenMQ [33] offers high quality, enterprise-ready messaging, it is the reference implementation for the JMS [34]. IronMQ [35] is an easy-to-use highly available message queuing service, it is built for distributed cloud applications with critical messaging needs. ElasticMQ [36]is a message queue system, offering an actor-based Scala and an SQS-compatible REST (query) interface. Cherami [37] is a distributed, scalable, durable, and highly available message queue system to transport asynchronous tasks.

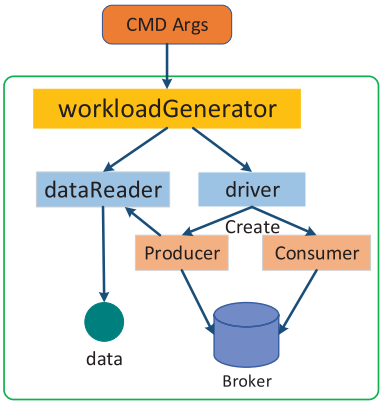


FIGURE 6. Test framework architecture.

V. PERFORMANCE EVALUATION RESULTS

In this section, we first build a test framework for fair performance comparison and then measure the performance features of these message queuing systems under this test framework.

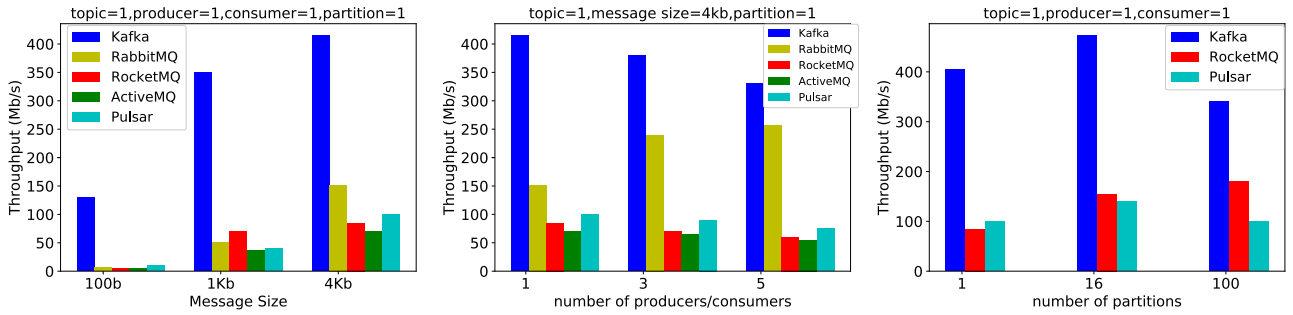
A. TEST FRAMEWORK

Most of the message queuing systems have provided their own built-in throughput/latency test tools. However, we do not use these tools to evaluate performance for the following two reasons: (1) The parameters that can be adjusted for each tool are different, so we cannot arbitrarily adjust the parameters as needed. (2) Each tool tests different metrics and measures them in different ways.

To ensure unity and fairness, we build our own test framework and write a throughput/latency test tool. FIGURE 6 shows the architecture of our test framework. In our framework, users should customize driver and workload in the form of command line arguments, which are paths of the configuration files. The *driver file* specifies the test messaging system and includes the basic configurations for this system. The *workload file* specifies different test conditions.

After reading the arguments, the *workloadGenerator* will create *dataLoader* and *driver*. According to the driver file, the driver will create producers and consumers connected to the specified Broker. The dataLoader is responsible for creating the Topic and reading the data to generate the message. The data is a randomly generated text file. We use Java to generate a random text file with arbitrary size. We provide datasets with different sizes in the data folder. If these data cannot meet the test requirements, users can customize the data generation.

After the producer and consumer are created, the producer will read the messages in dataLoader and send it to the Broker.



(a) Varying message size (b) Varying number of producers/consumers (c) Varying number of partitions

FIGURE 7. Throughput comparison.

The consumer will read it from the Broker. During this period, the workloadGenerator will record information such as the number of messages and timestamps, and periodically calculate the performance of the system. All the systems are tested under the same workload for fair comparison. The workload we used contain the fixed parameters that we will test. Users can change or add parameter configurations as needed.

Referring to the evaluation methods of [5], [11], [16], [38], in order to obtain correct and fair results without the effect of network, we perform benchmarking on a single machine, with the hardware configuration: 12 cores@3.6GHz, 16GB memory, and SSD for persistent storage. The versions of the tested message systems are listed as follows: Kafka 2.2.1, RabbitMQ 3.8.1, RocketMQ 4.5.1, ActiveMQ 5.15.0 and Pulsar 2.6.0. We use their default recommended configurations.

In our performance evaluation, we consider three factors: (i) message size. Message size has significant effect on the performance. (ii) number of producers/consumers. We vary the number of producers/consumers to test the performance under different levels of workload. (iii) number of partitions. We vary the number of partitions to evaluate the scalability. For each factor, we measure three different variants.

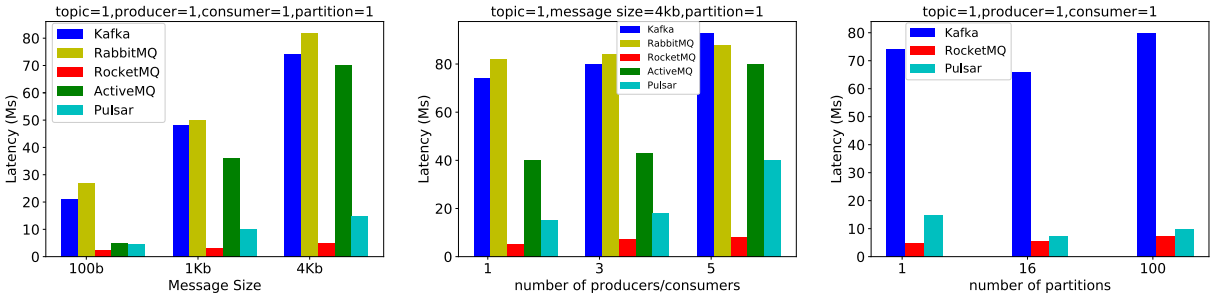
B. THROUGHPUT RESULTS

Considering the impact of different factors (e.g., the number of topics/producers/consumers/partitions and the message size), we test the throughput of the systems with different parameter settings. FIGURE 7a shows the throughput results when varying the message size and fixing the the number of topics/producers/consumers/partitions. With the growth of message size, the throughput of the systems increase gradually. These systems all use batch processing technology. That is, when the messages are accumulated to reach a threshold, they will be sent uniformly, thereby reducing transmission overhead and increasing throughput. Therefore, as the message size increases, the time waiting for batch processing decreases, and the throughput increases accordingly.

FIGURE 7b shows how the number of producers/consumers influences the throughput of these systems. In Kafka, a consumer subscribes to a topic, and a topic is consist of multiple partitions. There can be multiple consumers in a consumer group. At a certain moment, a message in a partition can only be consumed by a single consumer instance in the group, this is why the throughput is decreasing when the number of consumers grows. Ideally, the numbers of consumers and partitions are equal, i.e., a consumer corresponds to a partition for maximizing performance. In RocketMQ, Pulsar and ActiveMQ, all consumers in the same consumer group process the messages belonging to a topic in a Round-Robin manner. The throughput varies with different number of consumers as the number of partitions is fixed. RabbitMQ can support high concurrency because of the nature of Erlang language, it performs better with the increasing number of producers/consumers.

Generally speaking, increasing the number of partitions can increase concurrency, which in turn improves performance. Since there is no partition concept (or similar concept) in RabbitMQ and ActiveMQ, we only draw the results of Kafka, RocketMQ and Pulsar. FIGURE 7c shows the effect of the number of partitions. With the increase of partition number, the throughput of RocketMQ is increasing gradually. As we have analyzed in Section IV-C, RocketMQ stores all messages in the same physical file which is helpful when a large number of write/read threads exist. When the number of topics/partitions increases, the policy of deconcentrated storage of messages to disks will lead to disk IO competition to cause performance bottlenecks, so the the RocketMQ performance cannot grow linearly with the number of partitions. As for Kafka and Pulasr, they rely on Zookeeper to maintain the meta-information about partitions and replicas. However, due to the capacity and performance of Zookeeper, the number of brokers and partitions has upper limits, so the throughput will decrease once the number of partitions exceeds a threshold.

As these figures show, under various test conditions, Kafka outperforms other systems in all test cases. This can be attributed to Kafka's key optimization techniques as discussed in Section IV-A, e.g., zero-copy, disk sequential read/write, and data compression optimizations. Specially, (1) With zero-copy technology, Kafka can skip the copy of the user buffer and create a direct mapping of disk space and memory. The data is no longer copied to the user buffer, which



(a) Varying message size (b) Varying number of producers/consumers (c) Varying number of partitions

FIGURE 8. Latency comparison.

greatly improves its performance. (2) Kafka persists message records to disk, random read and write on disk can be really slow. Kafka adopts sequential read/write to improve the speed of disk I/O. Messages are appended to the end of a local disk file sequentially instead of random write operations, resulting in a significant increase in Kafka's write throughput. (3) If compressing each message, the compression rate can be quite low. Kafka employs bulk compression instead of compression for each individual message, as a result the transfer overhead is greatly reduced.

C. LATENCY RESULTS

We also test the latency of these message queuing systems with various parameter settings. Latency measures how long it takes for a message to be transmitted between endpoints, which is also a key metric for messaging systems. The latency results can be attributed to many factors, such as metadata processing, packet replication, memory access, and dequeue operations. FIGURE 8a shows the latency results of different systems when varying the message size and fixing the number of topics/producers/consumers/partitions. The latency of all systems increases with the increasing message size as the overhead of message transfer and disk I/O is increasing for larger messages. Kafka and RabbitMQ show longer latency results than the other systems, as Kafka and RabbitMQ employ batch processing to improve throughput at the expense of latency performance. In addition, RocketMQ distinguishes itself from other systems for its evident superiority in latency performance, which will be explained later.

FIGURE 8b shows the effect of the number of producers/consumers on the latency result. We fix the number of partitions and the number of topics both as 1, fix the message size as 4KB, and vary the number of producers/consumers from 1 to 5. With the number of producers/consumers grows, the concurrency overhead is increasing. This causes the increased latency as the figure shows. Similar to the latency results of previous experiment, we can see that Kafka and RabbitMQ show longer latency than the other systems, and RabbitMQ exhibits much better latency performance than the other systems.

FIGURE 8c shows the latency result when varying the number of partitions. We fix the number of topics/producers/consumers and the message size, and vary the number of partitions from 1 to 100. As there is no topic/partition concept in RabbitMQ and ActiveMQ, we omit their results. From the figure, we observe that as the number of partitions increased, the latency of Kafka and Pulsar first decrease and then increase. This is caused by the performance of Zookeeper and the disk competition. In addition, Kafka still exhibit very poor latency performance than others.

From these figures, we observe that RocketMQ exhibits marked superiority on the latency metric. RocketMQ's latency does not exceed 10ms, which is great in many application scenarios. To explore lower latency, RocketMQ has employed multiple optimization techniques to reduce the following latencies. (1) Reducing JVM Pause Latency. The JVM can produce many pauses during its operation, including GC (Garbage Collection), RedefineClasses, etc. By means of adjusting stack size, GC timing, and optimizing the data structure, RocketMQ can reduce the JVM pause latency significantly. (2) Reducing Lock Latency. Locking is widely used in the multi-threaded applications and can cause long thread waiting. RocketMQ uses optimistic lock to reduce the lock latency. PutMessageSpinLock is used by default to improve the lock unlocking efficiency for high contention workloads and to reduce the overhead of thread context switches. (3) Reduce Page Cache Latency. Page cache can prompt the read-write speed, but the writeback of dirty pages can lead to latency. RocketMQ employs memory predistribution, file preheating, and mlock system calls to reduce the page cache latency.

VI. DISCUSSION ON BEST-SUITED USE CASES

Our comprehensive analysis and test results provide a fair comparison of these message queuing systems. We summarize our comparison results in a radar chart as shown in FIGURE 9. Different systems have different pros and cons. These systems have different design philosophies and optimization techniques, and are competent in various scenarios. For example, Kafka is endowed with a higher throughput while RocketMQ distinguish itself for its superiority in latency performance. We can choose different message queuing systems according to application requirement. In this

(a) Kafka (b) RabbitMQ

(c) RocketMQ (d) ActiveMQ

(e) Pulsar

FIGURE 9. System comparison.

section, we offer several guidelines to choose these system candidates for different use cases.

A. REQUIRE FOR HIGH THROUGHPUT

Kafka is the best choice for users who need to do large-scale data collection and analysis. Its high throughput can support the collection of streaming data, and it can be used as the data source of many data analysis tools, such as Flink [23], Spark [22], Storm [39]. In general, Kafka is suitable for web site activity tracking, streaming data collection and monitoring, log merging, etc, and play the role of data source of big data processing frameworks. It is recommended that the number of consumers in the consumer group be the same as the number of partitions, and that the appropriate number of partitions should be set at each Broker to maximize performance.

B. REQUIRE FOR LOW LATENCY

If the message queuing system is used to process online business, which requires low latency for high quality of service, RocketMQ is preferred. Due to its high reliability and low latency, RocketMQ can be used in order transactions, recharge, messaging push, real-time analysis and many other applications. RocketMQ can support large amounts of topics and message accumulation so that it can also be used in complex business scenarios.

C. REQUIRE FOR MultiFunction

Users may need message queue to provide multiple functionalities to implement their application systems. In such a case, Pulsar is usually a good choice because it offers more functions than other message queuing systems, such as delay queuing and priority queuing. If Pulsar does not meet the requirements, it is recommended to select an eligible system according to TABLE 1.

VII. CONCLUSION

In this article, we comprehensively evaluate five popular message queuing systems. We compare Kafka, RabbitMQ, RocketMQ, ActiveMQ, and Pulsar from multiple dimensions, including production features, quality-of-service guarantee features, performance features, and functionality features. The comparison analysis result of these systems is summarized in TABLE 1. To fairly compare the throughput/latency performance, we design a test framework with a unified tool to emulate producer and consumer. Our test results show that Kafka has a higher throughput while RocketMQ has a lower latency. Therefore, we can choose different message queuing systems according to different application requirements. If the message queue is used to process online business, which requires low latency for quality of service, RocketMQ is preferred. If the message system is used to process massive amounts of messages, e.g., log processing, big data analysis, and stream processing, Kafka is preferred. In addition, Pulsar is appropriate if some special functionality is required.

REFERENCES

[1] D. Borthakur, "HDFS architecture guide," *Hadoop Apache Project*, vol. 53, nos. 1–13, p. 2, 2008.
[2] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon S3 for science grids: A viable solution?" in *Proc. Int. workshop Data-Aware Distrib. Comput.*, 2008, pp. 55–64.
[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[4] S. Shahrivari, "Beyond batch processing: Towards real-time and streaming big data," *Computers*, vol. 3, no. 4, pp. 117–129, Oct. 2014.
[5] T. Sebastian, "A comparison of data ingestion platforms in real-time stream processing pipelines," Malardalen Univ., Vasteras, Sweden, Tech. Rep. diva2:1440436, 2020.
[6] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. Int. Workshop Netw. Meets Databases*, vol. 11, 2011, pp. 1–7.
[7] 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.
[8] J. Yongguo, L. Qiang, Q. Changshuai, S. Jian, and L. Qianqian, "Message-oriented middleware: A review," in *Proc. 5th Int. Conf. Big Data Comput. Commun. (BIGCOM)*, Aug. 2019, pp. 88–97.
[9] B. Christudas, "Activemq," in *Practical Microservices Architectural Patterns*. Berlin, Germany: Springer, 2019, pp. 861–867.
[10] K. Ramasamy, "Unifying messaging, queuing, streaming and light weight compute for online event processing," in *Proc. 13th ACM Int. Conf. Distrib. Event-Based Syst.*, Jun. 2019, p. 5.
[11] J. Eriksson, "Comparing message-oriented middleware for financial assets trading," KTH School Technol. Health, Stockholm, Sweden, Tech. Rep. diva2:934286, 2016.
[12] Y. Liu, L.-J. Zhang, and C. Xing, "Review for message-oriented middleware," in *Int. Conf. Internet Things*. Berlin, Germany: Springer, 2020, pp. 152–159.
[13] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," in *Proc. 14th Int. Conf. Netw. Edu. Res.*, Sep. 2015, pp. 132–137.
[14] N. Nannoni, "Message-oriented middleware for scalable data analytics architectures," KTH School Inf. Commun. Technol., Stockholm, Sweden, Tech. Rep. diva2:813137, 2015.

[15] V. John and X. Liu, "A survey of distributed message broker queues," 2017, *arXiv:1704.00411*. [Online]. Available: http://arxiv.org/abs/1704.00411
[16] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proc. 11th ACM Int. Conf. Distrib. Event-Based Syst.*, 2017, pp. 227–238.
[17] S. Appel, K. Sachs, and A. Buchmann, "Towards benchmarking of AMQP," in *Proc. 4th ACM Int. Conf. Distrib. Event-Based Syst.*, 2010, pp. 99–100.
[18] J. Wagener, O. Spjuth, E. L. Willighagen, and J. E. Wikberg, "XMPP for cloud computing in bioinformatics supporting discovery and invocation of asynchronous Web services," *BMC Bioinf.*, vol. 10, no. 1, p. 279, Dec. 2009.
[19] M. Hemdi and R. Deters, "Using REST based protocol to enable ABAC within IoT systems," in *Proc. IEEE 7th Annu. Inf. Technol., Electron. Mobile Commun. Conf. (IEMCON)*, Oct. 2016, pp. 1–7.
[20] V. Wang, F. Salim, and P. Moskovits, "Using messaging over Websocket with stomp," in *The Definitive Guide to HTML5 WebSocket*. Berlin, Germany: Springer, 2013, pp. 85–108.
[21] C. N. Nguyen, J. Lee, S. Hwang, and J.-S. Kim, "On the role of message broker middleware for many-task computing on a big-data platform," *Cluster Comput.*, vol. 22, no. S1, pp. 2527–2540, Jan. 2019.
[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, nos. 10–10, p. 95, Jun. 2010.
[23] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, pp. 23–38, 2015.
[24] A. Bondarenko and K. Zaytsev, "Studying systems of open source messaging," *J. Theor. Appl. Inf. Technol.*, vol. 97, no. 19, pp. 5115–5125, 2019.
[25] B. R. Hiraman, C. Viresh M., and K. Abhijeet C., "A study of apache kafka in big data stream processing," in *Proc. Int. Conf. Inf., Commun., Eng. Technol. (ICICET)*, Aug. 2018, pp. 1–3.
[26] *Kafka Document*. Accessed: Sep. 2020. [Online]. Available: https://kafka.apache.org/081/documentation.html
[27] F. Junqueira and B. Reed, *ZooKeeper: Distribation Process Coordination*. Newton, MA, USA: O'Reilly Media, 2013.
[28] N. Garg, *Apache Kafka*. Birmingham, U.K.: Packt, 2013.
[29] G. Shapira, N. Narkhede, and T. Palino, *Kafka: The Definitive Guide*. Newton, MA, USA: O'Reilly Media, 2017.
[30] *Kafka vs. Apache Rocketmq Multiple Topic Stress Test Results*. Accessed: Sep. 2020. [Online]. Available: https://www.alibabacloud.com/blog/kafka-vs-rocketmq–multiple-topic-stress-test-results-69781
[31] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with BookKeeper," *ACM SIGOPS Oper. Syst. Rev.*, vol. 47, no. 1, pp. 9–15, Jan. 2013.
[32] P. Hintjens, *ZeroMQ: Messaging for Many Application*. Newton, MA, USA: O'Reilly Media, 2013.
[33] A. F. Klein, M. Stefanescu, A. Saied, and K. Swakhoven, "An experimental comparison of ActiveMQ and OpenMQ brokers in asynchronous cloud environment," in *Proc. 5th Int. Conf. Digit. Inf. Process. Commun. (ICDIPC)*, Oct. 2015, pp. 24–30.
[34] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java message service," Sun Microsyst., Santa Clara, CA, USA, Tech. Rep. 011801, 2002, vol. 9.
[35] *Ironmq: Elastic and Scalable Message and Event Handling*. Accessed: Sep. 2020. [Online]. Available: http://www.iron.io/products/mq
[36] *Elasticmq: A Fully Asynchronous, Akka-Based Amazon SQS Server*. Accessed: Oct. 2020. [Online]. Available: https://softwaremill.com/elasticmq-asynchronous-akka-based-amazon-sqs-server
[37] *Cherami: Uber Engineering's Durable and Scalable Task Queue in Go*. Accessed: Oct. 2020. [Online]. Available: https://eng.uber.com/cherami-message-queue-system/
[38] X. J. Hong, H. Sik Yang, and Y. H. Kim, "Performance analysis of RESTful API and RabbitMQ for microservice Web application," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2018, pp. 257–259.
[39] M. Hussain Iqbal, SZABIST, and T. Rahim Soomro, "Big data analysis: Apache storm perspective," *Int. J. Comput. Trends Technol.*, vol. 19, no. 1, pp. 9–14, Jan. 2015.

GUO FU is currently pursuing the master's degree with Northeastern University, China, where he is also pursuing the M.S. degree in computer science. His research interests include databases and stream processing.

YANFENG ZHANG (Member, IEEE) received the Ph.D. degree in computer science from Northeastern University, China, in 2012. He is currently a Professor with Northeastern University. His research interests include distributed systems and big data processing. He has published many articles in the above areas. His article in SoCC 2011 was honored with "Paper of Distinction".

GE YU (Senior Member, IEEE) received the Ph.D. degree in computer science from the Kyushu University of Japan, in 1996. He is currently a Professor with Northeastern University, China. He has published more than 200 articles in refereed journals and conferences. His current research interests include distributed and parallel database, OLAP and data warehousing, data integration, and graph data management. He is a member of the ACM and the CCF.