
**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE**
**CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA**
**UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO**
**CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**RONAN DE ARAÚJO SOUZA**

# PERFORMANCE ANALYSIS BETWEEN APACHE KAFKA AND RABBITMQ

**CAMPINA GRANDE - PB**

**2020**

**RONAN DE ARAÚJO SOUZA**

# PERFORMANCE ANALYSIS BETWEEN APACHE KAFKA AND RABBITMQ

**Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.**

**Orientador: Professor Dr. Thiago Emmanuel Pereira da Cunha Silva.**

**CAMPINA GRANDE - PB**

**2020**


S729p Souza, Ronan de Araújo.
Performance analysis between Apache Kafka and RabbitMQ. / Ronan de Araújo Souza. – 2020.

11 f.

Orientador: Prof. Dr. Thiago Emmanuel Pereira da Cunha Silva.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Distributed systems. 2. Apache Kafka. 3. RabbitMQ. I. Silva, Thiago Emmanuel Pereira da Cunha. II. Título.

CDU:004.6(045)

**Elaboração da Ficha Catalográfica:**

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

**RONAN DE ARAÚJO SOUZA**

# PERFORMANCE ANALYSIS BETWEEN APACHE KAFKA AND RABBITMQ

**Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.**

## BANCA EXAMINADORA:

**Professor Dr. Thiago Emmanuel Pereira da Cunha Silva**
**Orientador – UASC/CEEI/UFCG**

**Professor Dr. Francisco Vilar Brasileiro**
**Examinador – UASC/CEEI/UFCG**

**Professor Dr. Tiago Lima Massoni**
**Disciplina TCC – UASC/CEEI/UFCG**

**Trabalho aprovado em: 2020.**

**CAMPINA GRANDE - PB**

# Performance analysis between Apache Kafka and RabbitMQ

Ronan de Araújo Souza
ronan.souza@ccc.ufcg.edu.br

Thiago Emmanuel Pereira
temmanuel@computacao.ufcg.edu.br

## ABSTRACT

This paper aims to compare two of the most used publish/subscribe systems: Apache Kafka and RabbitMQ.

Publish/Subscribe (pub/sub) is a pattern that is used to enable asynchronous communication between different applications, it usually is implemented in the form of a message queue that holds the content sent by producers and delivers it to consumers.

With Apache Kafka and RabbitMQ being the most common pub/sub platforms, each one having its characteristics, the question of how to properly compare them and how to choose the best fit for a specific application always comes up to mind.

To answer this question, we define the core functionalities of pub/sub systems and compare how each platform implements it, as well as present the results of a benchmark to measure quantitative metrics and point out distinct aspects of each one. In the end, we list the main use cases for publish/subscribe systems and which tool is best suited based on all results previously obtained.

## Keywords

Distributed systems, publish/subscribe platforms, Apache Kafka, RabbitMQ

## 1. INTRODUCTION

With distributed solutions being increasingly adopted from startups to traditional business, the need for tools to work as middlewares that provide decoupled and asynchronous communication between multiple services geographically distributed.

From this scenario, publish/subscribe platforms such as Apache Kafka and RabbitMQ are widely adopted to provide reliable, efficient, and safe message exchange between parts.

Although both platforms have the same primary purpose, each one also has particular features that could either be important or not feasible for a production environment. With different characteristics and approaches, which one to adopt?

In this paper, we will first give a background description of the publish/subscribe paradigm and list its functionalities (Section 2). After, in Section 3 and 4, we will describe Kafka and RabbitMQ implementation. Then, we provide qualitative (Section 5) and quantitative (Section 6) comparisons of common features for both platforms. In Section 7, we list the important features of each tool. In section 8, we discuss the main differences between the Philippe's and Kyumars' [1] work and the results obtained now. Finally, in Section 9, we list the best-suited use cases for Kafka and RabbitMQ, and in Section 10 we finish with considerations about the changes that happened between the experiment.

The methodology used to compare the platforms is a reimplementation of Philippe Dobbelaere and Kyumars Sheykh Esmaili [1] work, reimplementation makes it necessary due to the release of new features for each platform that would impact the final result.

## 2. BACKGROUND

In this section, we highlight the main concepts of the pub/sub paradigm. The concepts raised here will be used further to the comparisons between both platforms.

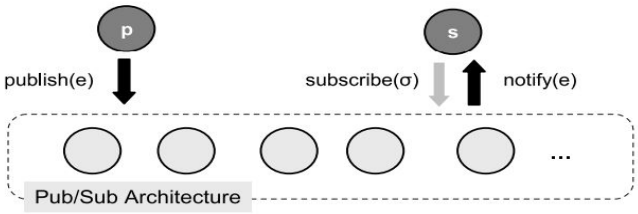


*Figure 1 : High-level interaction model of a publish/subscribe system with its clients (p and s indicate a generic publisher and a generic subscriber, respectively, while e is the message and σ is the subscription) [15] .*

### 2.1 Concepts

Publish/subscribe (henceforth, pub/sub) is a messaging pattern that uses a middleware responsible for exchanging messages between clients. The clients of pub/sub systems are divided according to their role into publishers, which act as producers of information, and subscribers, which act as consumers of information [4]. Clients are not required to communicate directly among themselves; instead, they have levels of decoupling between services that are vital for some kinds of distributed applications. These levels are:

#### *2.1.1 Entity decoupling*

Entity decoupling is due to neither publishers nor subscribers being aware of the existence of the other. It happens because they are connected only with the platform, so if the subscriber(s) are offline, the publishers will keep sending messages to the platform. The same occurs in the opposite direction;

#### *2.1.2 Time and synchronization*

Time and synchronization decoupling happens because the parts do not need to be active at the same time, and they will not be blocked while the other part is offline. E.g. if the publisher sends a message and there are no subscribers to receive that message, the

platform will keep that to be further delivered to subscribers (or delete, depending on the configuration).

#### *2.1.3 Message Routing*

Message routing determines how, and if, the produced messages will be delivered to the consumers. There are two main types of routing that are used by Kafka and RabbitMQ; they are based in:

##### *2.1.3.1 Content*

Which means that the consumers can filter the messages they want to receive by its content. Because of the need for processing at the consuming, this kind of routing could be more expensive than the next one;

##### *2.1.3.2 Topic*

Where the message is sent to a topic(s) defined by the publisher when they are produced, so consumers can subscribe to topics to have the messages already filtered.

### 2.2 Quality of Service Guarantees

The lack of a direct producer/consumer relationship makes the definition and enforcement of any end-to-end QoS policy very hard [4]. Here we will describe the main QoS guarantees that pub/sub systems must have. Using the same approach of [1], we divided the guaranties into groups, as described below:

#### *2.2.1 Correctness*

Correctness behavior can be divided into two kinds of guarantees:

##### *2.2.1.1 Delivery Guarantees*

- *At most once*

At most once guarantees that the system will deliver no duplicate messages, but in case of a packet loss, some message can be lost;

- *At least once*

At least once assures that if some packet is lost, another one will be sent. It means that no message will be lost, but in case of a false positive the consumer may receive the same message twice;

- *Exactly once*

Exactly once is the most expensive one, because of the need for two-phase commits to assure no loss and no duplication.

##### *2.2.1.2 Ordering Guarantees*

- *No ordering*

No ordering no have any guarantee about message ordering, but normally can achieve a better performance

- *Partitioned Ordering*

Partitioned ordering means that some partitions are ordered, but when it comes to multiple partitions, there is no guarantee. It costs more than the previous one, but it allows the system to scale horizontally and if there is a need for some order to be followed it can be kept into the same partition.

- *Global Ordering*

Global ordering means that every message will be delivered in the same order of production. It needs to keep every messaging channel synchronized, which is very expensive in a distributed system.

#### *2.2.2 Availability*

Availability is the ability of the system to keep uptime. It could be measured as the percentage of time that the system is available over the measurement period. It works to estimate the future performance of this system.

#### *2.2.3 Transactions*

Transactions are used to encapsulate messages before sending them. It is used to reduce the use of network resources, or when it's known that the consumer needs all the messages together to process them.

#### *2.2.4 Scalability*

Scalability is the system's ability to increase his capability to properly handle the workload. A pub/sub system could scale to be able to process more messages faster, or deal with a major number of clients (consumers and producers).

#### *2.2.5 Efficiency*

Efficiency is the capacity to achieve the desired result with fewer resources. In the experiment described in section 6, we will use the following metrics to measure the efficiency of both systems.

##### *2.2.5.1 Latency*

Latency is the time spent to a data packet to travel from a point to another. In this case, from the producer(s) to the consumer(s).

##### *2.2.5.2 Throughput*

Throughput the number of packets (or bytes) per time unit that can be transported between producers and consumers. Contrary to latency, throughput can easily be enhanced by adding additional resources in parallel [1].

## 3. APACHE KAFKA

Apache Kafka was originally developed by LinkedIn but in 2011 it was donated to Apache Foundation and has been maintained by them since.

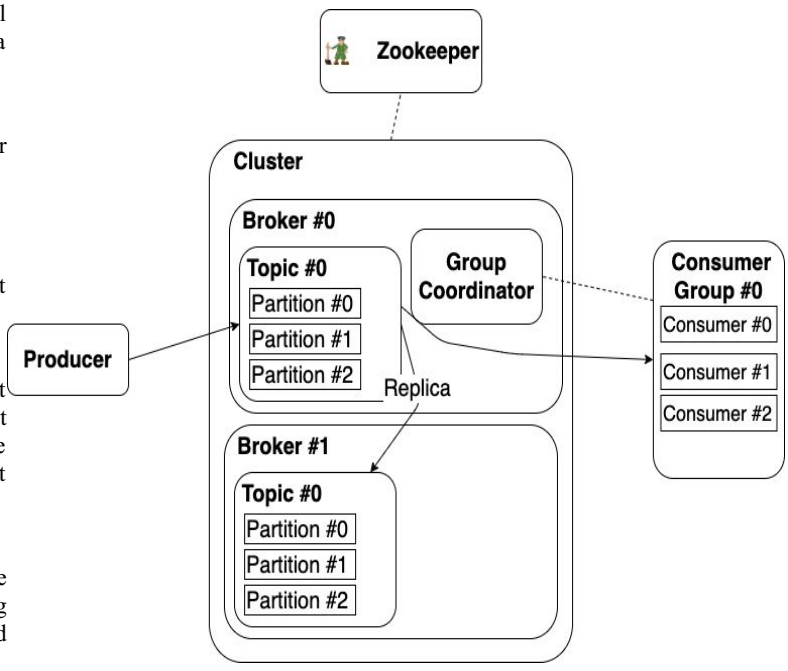


*Figure 2: Kafka Architecture*

First, it is important to know that Kafka relies on Zookeeper to manage clusters, topics, and partitions. So it's necessary to have a Zookeeper instance running to deploy a Kafka Cluster.

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services [10].
When a producer sends a message to a topic, it's stored into a partition in this topic that will be consumed for all the consumer groups that are listening to this specific topic as is visible in figure 1. At first sight, it is not clear why consumers are grouped and topics are divided into partitions, let's have a look:

Partitions are the topic divisions to enable horizontal scaling of a topic, so a topic could have partitions (and their replicas) over different brokers in the cluster. As shown in Figure 2, a partition is an ordered log of messages which makes a topic partial ordered;

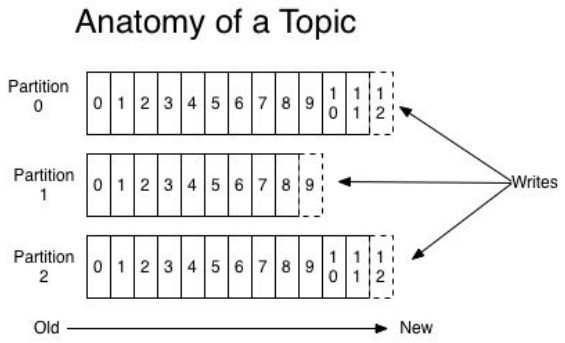


*Figure 3: Anatomy of a Topic [16]*

As above, when it is necessary to horizontally scale a consumer is simple. Consumers with the same consumer group ID make the Kafka group coordinator divide the topic partitions between all the consumers[1].

With the partitions/consumer group approach, Kafka makes possible horizontal scaling with partial ordering and no duplication in the ideal scenario - to increase performance, Kafka has an option to commit the consumed offset periodically. Once it is not necessary to commit the offset after each consumption, the process becomes faster. But, on the other hand it only guarantees at least once delivery due to the possibility that a crash happens just before the consumer commits the current offset causing some message duplication.

# 4. RABBITMQ

RabbitMQ is an open-source message broker developed by Rabbit Technologies and is now maintained by Pivotal Software. RabbitMQ is an Erlang implementation of Advanced Message Queuing Protocol (AMQP), which is a protocol for message-oriented communication that relies on **message queuing** to store messages coming from **exchanges** and deliver them to consumers.

[1] It is worth mentioning that if there are more consumers than partitions some of them will be idle.

In section we first give an explanation about the RabbitMQ core concepts and describe a message flow since the publishing from the producer until the reception of the consumer in 4.1. Then, in 4.2 we describe all the different ways that the message can be routed to a queue.

## 4.1 Core Concepts

In RabbitMQ, exchanges are responsible for getting messages from producers and depending on its type, choose which queue will receive the message, to choose the correct queue it uses bindings, that specificities rules and the criteria to define a route between an exchange and a queue, that is where messages are stored until they are handled by consumers. There is also the routing key, a message attribute used by some exchanges to select which queue will receive the message.

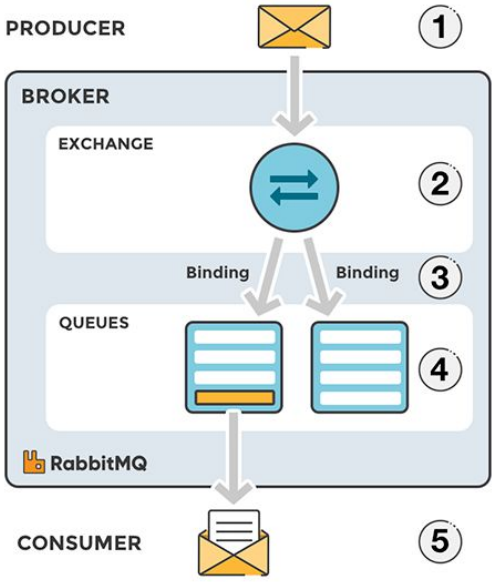


*Figure 4: Message flow in RabbitMQ [5]*

As seen in figure 4, the message flow in RabbitMQ has 5 steps, let's look at it:

1. Producer sends a message to a RabbitMQ broker, it is first delivered to the exchange;
2. Exchange, depending on its type, will choose which queue will receive the message;
3. If the message is compatible with some biding, the exchange will send it to the queue, instead, the message is lost;
4. The queue receives the message and keeps it until some consumer handle it;
5. The consumer finally receives the message.

## 4.2 Exchanges types

- *Direct*

The message will be sent to the queue that the binding key matches exactly the routing key.

- *Headers*

The exchange will consider the message headers as a routing key.

- *Fanaut*

Similar to a broadcast, the message is sent to all queues binded.

- *Topic*

Is quite similar to the direct type, but in this case, the binding key could be a regular expression.

It's worth mentioning that queues keep messages in memory when it's possible, putting them on disk only when necessary. But if the persistent option is selected, all messages will be stored on disk possibly causing higher latency.

# 5. QUALITATIVE COMPARISON

In this section we list all the primary features that both platforms provide and the approach of each implementation, as well as possible drawbacks/advantages.

## 5.1 Time decoupling

Both systems can store messages to be consumed later, but each one deals with it in a different way.

Kafka is designed to handle a considerably high amount of data and can scale better.

RabbitMQ stores all new messages in DRAM memory and once there is no more memory available it starts storing in the disk which can degrade the performance.

## 5.2 Routing Logic

Once that RabbitMQ inherits routing logic from AMQP with the exchanges approach, it is already very flexible in terms of logic. Not enough, RabbitMQ has an API making possible making the routing logic even more customizable.

On the other hand, Apache Kafka has only the topic-based logic natively. **KsqlDB** (see section 7.1.4) raised the possibility to query from message parameters, but it adds a higher complexity level.

## 5.3 Delivery Guarantees

Both Kafka and RabbitMQ have the guarantees of at least once, which means that all messages got delivered but some messages could get duplicated; at most once guarantee it makes sure that there is no duplication but in case of failure some messages could be lost. Kafka also has a specific scenario where it is possible to guarantee exactly one message is delivered, that will explore in 7.1.

### *5.3.1 Apache Kafka*

Same as RabbitMQ, it is possible to guarantee message durability and acknowledgment. For durability, it is possible to replicate partitions through many brokers into the cluster, and when the leader goes down, one follower becomes a leader and keeps the data available, but for this work properly, all the replicas should be synchronized. Kafka has the concept of In Sync Replicas (ISR). Each replica can be in or out of sync. In sync means that they have been up-to-date with the leader within a short period (the last 10 seconds by default) [6].

The producer can define which kind of acknowledgment it wants to receive from the broker, that could be:

- No acknowledgment, fire and forget. Acks=0.
- The leader has received and processed the message. Acks=1
- The leader and all In Sync Replicas have received and processed the message. Acks=All

### *5.3.2 RabbitMQ*

It's possible to guarantee message durability and message acknowledgment. Durability, that is, not losing the message when the broker fails is achieved using quorum queues [6], which enables high availability of the data.

For acknowledgment is possible to set up a publisher to wait for the confirmation message, that could be a *basic.ack* meaning that the message was successfully received and processed or *basic.nack* when something happened while sending or processing the message. Waiting for an ack after each message could seriously degrade the throughput, so it's possible to set up the producer to send a steady stream of messages until it reaches a limited number of unacknowledged messages, then it pauses and waits for the confirmation.

## 5.4 Ordering Guarantees

RabbitMQ is possible to have a queue fully ordered even for retransmitted messages, once is used a single AMQP channel.

For Kafka it is not possible to guarantee that a topic is ordered, however is possible to achieve that into each partition.

## 5.5 High Availability

Both platforms provide availability using replication.

For Apache Kafka is necessary to define the replication factor in the topic creation. It will replicate (replication factor times) each partition in a different broker in the cluster. It's worth mentioning that your replication factor should be at least the same size as the available brokers.

RabbitMQ only applies high availability to mirrored or quorum queues [6], while classic queues will not be replicated.

## 5.6 Multicast

When comes to the need of sending the same message to multiple clients each platform deals in a different way:

RabbitMQ provides multicast by creating a queue for each consumer, which depending on the number of consumers could highly increase the number of bindings to support the individual queues.

Kafka is completely transparent at the server-side, this is due to the fact that the message is delivered once to each partition replica and there is a consumer offset coordinator responsible to manage the offset of each consumer.

## 5.7 Scalability

Scaling a RabbitMQ cluster is well supported and can be done online (there will be no downtime). For adding new nodes, those will be able to become master for new queues and will accept connections to publish/consume to/from any queue; for removing existing nodes, it's quite similar, just being necessary to run a forget_cluster_node command to remove a node from the cluster.

In Apache Kafka, the dynamic scale is not completely transparent to the consumer, since there is a mapping for consumers to partitions in a consumer group. For adding new nodes, it is necessary to define which existing partitions will be replicated to the new broker, but the process can be done with no downtime. To

remove existing brokers first is necessary to redistribute all the present partitions on this node to existing ones before it can be done.

# 6. QUANTITATIVE COMPARISON

In this section, we will compare the performance of each platform from two essential metrics: **Throughput** and **latency**. The results presented in this document are native from a benchmark developed by Alok Nikhil and Vinoth Chandar [9], supported by Confluent to compare RabbitMQ, Apache Kafka, and Apache Pulsar. For more detailed information, it's possible to see all the benchmark results in [11].

## 6.1 Test Environment

The experiment was executed using AWS EC2 instances **i3en.2xlarge** (with 8 vCores, 64 GB RAM, 2 x 2,500 GB NVMe SSDs). For these tests, four instances were used to produce the workload, three nodes to host the Kafka/RabbitMQ brokers (Kafka also required three nodes for Apache Zookeeper) and one node to monitor the environment. The systems versions utilized in the tests were 2.6 for Apache Kafka and 3.8.5 for RabbitMQ.

To run the benchmark was the OpenMessaging Benchmark Framework with changes to add the RabbitMQ driver, once this option is not available in the framework yet.

## 6.2 Latency

With the popularity of pub/sub systems in stream processing and event-driven architecture, realtime delivery and low end-to-end latency is a decisive factor when it comes to choosing one of those systems.

### *6.2.1 Methodology*

To measure the end-to-end latency it was defined as the highest stable throughput for each system based on previous runs that showed a throughput at **200k messages per second for Apache Kafka** and **30K messages per second for RabbitMQ**. The considerable difference in the defined throughput due to the CPU bottleneck faced by RabbitMQ, this problem is evident in the 6.3 section.

Both systems were set up to high availability, meaning that RabbitMQ used mirrored queues; to achieve the best performance was defined that Kafka *fsync* config turned off and RabbitMQ does not persist messages on disk.

### *6.2.2 Results*

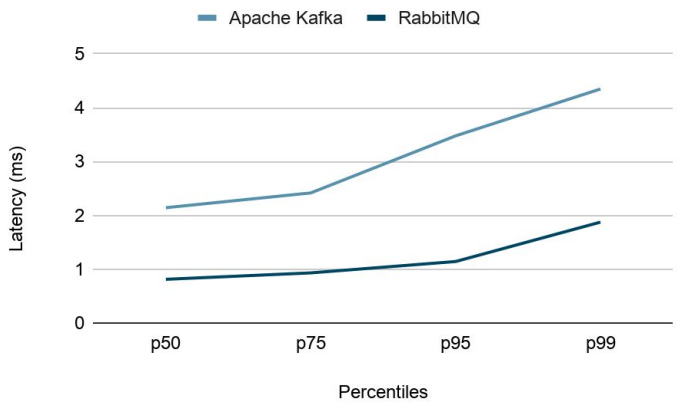


*Chart 1: End-to-end latency between Apache Kafka and RabbitMQ [11]*

It's possible to notice from the chart above that RabbitMQ got better results than Kafka. Even though under a limitation that prevents RabbitMQ from horizontal scale without a CPU bottleneck that in this specific scenario it's with mentioning that the max latency of RabbitMQ was **27.572 ms** against **94.178 ms** from Apache Kafka, which means more than 3 times better.

This scenario changes completely with bigger throughput, with p99 reaching 2 seconds in RabbitMQ at 38K messages per second.

## 6.3 Throughout

This test aimed to measure the *peak stable throughput,* which is: The highest producer throughput average at which consumers can handle without an ever-growing backlog.

### *6.3.1 Methodology*

For this test the replication factor 3 was defined, which means that all messages were replicated 3 three times across the nodes for high availability, and to enable a higher throughput both systems used a batch with 1 MB where each message was 1 KB size.

For Kafka was created one topic with 100 partitions, while RabbitMQ had a single direct exchange linking to 24 queues (since each queue required a dedicated core, was used 3 brokers each one with 8 vCPUs), and in this case, the benchmark framework used a round-robin to generate message keys to enable the exchange equally route to all queues.

### *6.3.2 Results*

Given the previously described scenario, Apache Kafka presented the highest throughput with the following metrics:

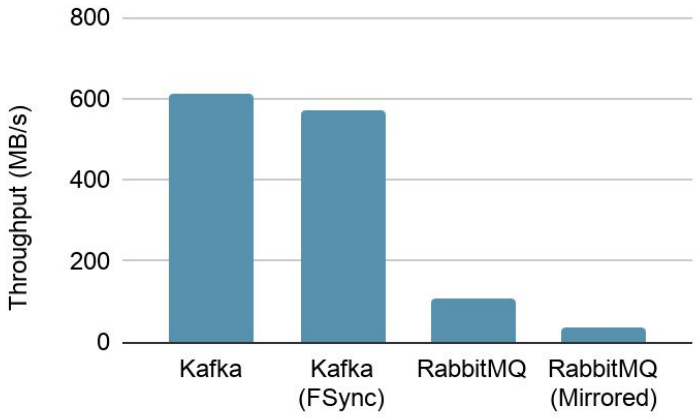


*Chart 2: Peak Stable Throughput with four producers and four consumers [11]*

To test Kafka's throughput performance, it was chosen two scenarios, using *fsync* that calls the *fsync* system call for each message to write the data in the disk before acknowledging the producer, and as expected it would have a degradation compared with the scenario with no *fsync*.

RabbitMQ tested two scenarios, the first one with no replication, while the second used mirrored queues to assure the availability of the message. It was noticed that RabbitMQ does not handle well with an overhead of replication, getting a CPU bound during the workload which severely degrades the performance.

# 7. DISTINCT FEATURES

## 7.1 Apache Kafka

### *7.1.1 Long Term Storage*

Don't fear the filesystem![2] Kafka relies heavily on the filesystem for storing and caching messages, and due to the way it's done - linear writes - it takes the best from disk, so it enables to store huge amounts of data. Each topic has a retention time that is used to purge messages older than that (or when the topic's disk quota is exceeded).

### *7.1.2 Kafka Connect*

Kafka Connect is a reliable open-source framework for connecting Kafka with external systems such as databases, key-value stores, search indexes, and file systems[7]. Connect runs with streaming and batch-oriented and it's a solution that reduces the development time in some cases.

### *7.1.3 Kafka Streams*

Kafka Streams is a lightweight client library to perform data processing, is an interesting option because it is fault-tolerant, elastic (client-side), equally viable for different scenarios (from a local test to a production cluster). Kafka stream is available in Java and Scala.

### *7.1.4 KsqlDB*

KsqlDB is an event streaming database, that is, is a particular kind of database to develop processing applications. With ksqlDB is possible to, for example, query from a topic filtering by some parameter using SQL syntax.

## 7.2 RabbitMQ

### *7.2.1 AMQP*

RabbitMQ is an open-source implementation of a standardized protocol (AMQP), and because of that, there is a higher level of similarity between other platforms based on the same protocol, which could be beneficial in a case of substitution, for example.

### *7.2.2 Message TTL*

A "time to live" could be essential in some real-time scenarios, where the message delivery could be nonsense after some time.

### *7.2.3 Publisher Flow Control*

RabbitMQ can stop publishers from sending messages, in order to keep the rate of messages being received to avoid a server being overwhelmed.

### *7.2.4 Message Prioritization*

RabbitMQ has priority queues, where publishers then can publish prioritized messages using the priority field (between 0 and 255) on the message body, larger numbers indicate higher priority. It's with mentioning that there is some in-memory and on-disk cost per priority level per queue. There is also an additional CPU cost, especially when consuming, so you may not wish to create huge numbers of levels[3].

### *7.2.5 UI and monitoring tools*

It comes with an easy-to-use interface attached that allows the user to monitor connections, queues, exchanges, clustering, resource consumption in a self-explanatory dashboard.

# 8. DISCUSSION

From the 2017 experiment until now, there have been some releases for both Apache Kafka and RabbitMQ, more specifically Kafka comes from 0.10.0.1 to 2.6 and RabbitMQ from 3.5.3 to 3.8.5. In this section, we aim to discuss the main features added to both platforms, as well as the differences found between both benchmarks.

## 8.1 Apache Kafka

For Apache Kafka, some important features were released in the last years, especially from Confluent that is an event Streaming Platform based on Kafka. One noteworthy feature is the transactions that were added in the 1.0.0 release. Transactions are specially util in cases of, for example, financial institutions use stream processing applications to process debits and credits on user accounts. In these situations, there is no tolerance for errors in processing: we need every message to be processed exactly once, without exception [14].

Another considerable feature is the ksqlDB, which is an event streaming database, for more information, see section 7.1.4.

## 8.2 RabbitMQ

For RabbitMQ, on the 3.6.0 release was added the Lazy queues, which attempts to move messages to disk as early as practically possible. This means significantly fewer messages are kept in RAM in the majority of cases under normal operation. This comes at a cost of increased disk I/O [3]. It differs from the default queue approach, that tries to keep in-cache message data whenever is possible.

But the most important feature since then is the **Quorum queues.** This new kind of queue comes to improve the model

synchronization and consequently the performance of mirrored queues without losing any of the high availability present on that. Quorum Queues are a kind of mirrored queues that use the Raft Consensus Algorithm [13] to replicate messages through the cluster. One message is acknowledged when a quorum master and a defined number of followers nodes receive it, assuring the high availability of the message. This also solves some problems from the classic mirrored queues such as when a node goes down and gets back online, it's not necessary a whole synchronization with the master, which its mirrored queues were a blocking process that would keep the entire queue unavailable. For more about quorum queues, see [12].

## 8.3 Quantitative Experiment

Before any further, it is essential to say that both experiments were carried out in different scenarios, so some considerations may be done. That being said, for latency matters, results seemed quite near, with both latency quite similar and RabbitMQ showing a small advantage.

Due to the fact of a high replication factor that caused a CPU bottleneck to RabbitMQ, which severely impacted the benchmark results, a proper comparison is not possible.

# 9. PREFERRED USE CASES

## 9.1 Apache Kafka

### *9.1.1 Analytics*

It's known that the primary use case of Apache Kafka was to track website activity, collecting all events generated by users, storing and processing them afterward. With big companies around the world with applications used by millions of people, it's clear the need for tools that can handle high throughputs. Another important factor is the accessibility possible because of Kafka connect, making it easy to create streams from Kafka to big data storage systems such as Elasticsearch and Spark.

### *9.1.2 Realtime*

Kafka is a distributed system with high-throughput, which is essential for systems with huge amounts of data that deals with real-time processing. Because of that, systems like Spotify and Shopify use Kafka to publish data in real-time.

### *9.1.3 Event Sourcing*

Event source is a design that captures all changes to an application state as a sequence of events[10]. With Kafka's support for storing large amounts of data, it's the best choice for applications in this scenario. Nubank and Wildlife Studios are heavy users of Apache Kafka in this scenario.

## 9.2 RabbitMQ

### *9.2.1 Middleware in Microservices Architecture*

As aforementioned, one popular application for pub/sub systems is providing communication between microservices. RabbitMQ is a better option for this scenario for some reasons: The First one is that normally messages do not need to be stored for a long period, which is a RabbitMQ's characteristic; It's also normal that each message will be consumed once, another property that is better handled by RabbitMQ.

Companies like Bloomberg and Parkster use RabbitMQ as middleware in their microservices architecture.

# 10. CONCLUSION

This paper aimed to reimplement a framework comparison between Apache Kafka and RabbitMQ that was previously established a couple of years ago by Philippe Dobbelaere and Kyumars Sheykh Esmaili. But beyond that, it also shows how both platforms worked to improve known issues and develop new features to adapt itself to the new market needs.

In terms of performance, Apache Kafka showed to be more scalable than RabbitMQ in global scenarios, where huge throughputs should be handled by many brokers with high availability and deliver it at considerably low latencies.

On the other hand, for some scenarios RabbitMQ showed to be a no-brainer choice, for instance when there is a need for global ordering with a befitting throughput.

Another important factor that should be noticed is the commitment of Apache Kafka to add new features to its toolbox. Especially from Confluent, which is a company created by some Kafka founders and provides a complete platform for streaming data, with cost planning for cloud resources and 24/7 support. Features like Kafka connect that provides a framework with connectors to external services in a really easy way, or even KsqlDB, that is an event streaming database that between other important features it added the possibility of filtering from the data content, that previously Kafka wasn't able to offer.

With the peculiarities of each system, a decision to each one be used should rely on different aspects as mentioned in the previous sections, but not only that. An aspect that should be considered is the adaptability of new features as shown in the last paper, where Kafka together with Confluent developed a toolkit to support new market demands, and on the other hand RabbitMQ making use of AMQP protocol which makes a change to another AMQP based system a smoother change.

# 11. REFERENCES


[1] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17). Association for Computing Machinery, New York, NY, USA, 227–238. DOI: https://doi.org/10.1145/3093742.3093908

[2] Apache Kafka: https://kafka.apache.org/documentation

[3] RabbitMQ: https://www.rabbitmq.com/documentation.html

[4] A. Corsaro et al. Quality of Service in Publish/Subscribe Middleware. Global Data Management, 19(20):1–22, 2006.

[5]CloudAMQP - RabbitMQ for Beginners: https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html

[6] RabbitMQ vs Kafka Part 4 - Message Delivery Semantics and Guarantees: https://jack-vanlightly.com/blog/2017/12/15/rabbitmq-vs-kafka-part-4-message-delivery-semantics-and-guarantees

[7] Confluent: https://docs.confluent.io/

[8]ksqlDB: https://docs.ksqldb.io/en/latest/?_ga=2.263364876.1243017561.1604802691-2027677595.1604802691

[9] Alok Nikhil and Vinoth Chandar, 2020. Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest? https://www.confluent.io/blog/kafka-fastest-messaging-system/

[10] Apache Zookeeper docs: https://zookeeper.apache.org/doc/r3.6.2/index.html

[11] OpenMessaging Benchmark Results: https://github.com/confluentinc/openmessaging-benchmark/tree/master/blog/results

[12] RabbitMQ 3.8 Feature Focus - Quorum Queues: https://www.cloudamqp.com/blog/2019-03-28-rabbitmq-quorum-queues.html

[13]The Raft Consensus Algorithm: https://raft.github.io/

[14] Transactions in Apache Kafka: https://www.confluent.io/blog/transactions-apache-kafka/

[15] A. Corsaro et al. Quality of Service in Publish/Subscribe Middleware. Global Data Management, 19(20):1–22, 2006.

[16] Lawlor, Brendan & Lynch, Richard & Mac Aogáin, Micheál & Walsh, Paul. (2018). Field of genes: using Apache Kafka as a bioinformatic data repository. GigaScience. 7. 10.1093/gigascience/giy036.

## About the authors:

**Ronan de Araújo Souza** is a senior Computer Science student at UFCG that currently works as a Site Reliability Engineer intern at Wildlife Studios.

**Thiago Emmanuel Pereira** is a Computer Science Professor at UFCG that works in the Distributed Systems Laboratory (LSD)