DDS-PROJECT

Di Felice Gianmarco, Lamanna Pietro, Muserra Davide

April 5, 2023

1 Introduction

The aim of the project was to carry out a comparison and a performance evaluation of the following solutions, that implement the publisher-subscriber message pattern:

- ***ActiveMQ***: it is an open-source message broker that provides a messaging system for distributed applications. It is designed to implement the Java Message Service (JMS) API, which is a standard messaging API for Java applications. ActiveMQ supports various messaging protocols such as JMS, AMQP, MQTT, and WebSocket. It is based on a publish-subscribe messaging model, where producers can publish messages to a topic and consumers can subscribe to that topic to receive messages. ActiveMQ also supports features such as message filtering, message persistence, and message grouping. It is written in Java and is available under the Apache License.
- ***RabbitMQ***: it is a message broker that provides a messaging system for distributed applications. It is built on top of the Advanced Message Queuing Protocol (AMQP) and provides support for other messaging protocols such as MQTT and STOMP. RabbitMQ is written in Erlang and provides a distributed architecture that allows it to scale horizontally. RabbitMQ uses a publish-subscribe messaging model, where producers can publish messages to a queue and consumers can consume messages from that queue. RabbitMQ also supports features such as message routing, message priority, and message persistence. It is available under the Mozilla Public License.
- ***Kafka***: Kafka is a distributed streaming platform that is designed for building real-time data pipelines and streaming applications. It is used to process large volumes of data in real-time and is built on top of a publish-subscribe messaging model. Kafka is written in Java and Scala and is available under the Apache License. Kafka provides a distributed architecture that allows it to scale horizontally and supports features such as fault-tolerant data replication, data retention, and data compression. Kafka is often used with other distributed systems such as Apache Spark and Apache Storm for stream processing.

  In summary, ActiveMQ, RabbitMQ, and Kafka are all messaging technologies that provide a messaging system for distributed applications. ActiveMQ and RabbitMQ are more traditional message brokers, while Kafka is designed specifically for real-time data streaming and processing. Each technology has its own strengths and use cases, and the choice between them depends on the specific needs of the application.

2 Comparison among the three tecnologies

It follows some key differences among ActiveMQ, RabbitMQ and Kafka:

- ***Messaging Model***: ActiveMQ and RabbitMQ use a traditional publish-subscribe messaging model where producers publish messages to a queue or topic and consumers consume messages from that queue or topic. Kafka, on the other hand, uses a distributed log-based messaging model where producers write messages to a log and consumers read messages from that log.
- ***Protocol Support***:ActiveMQ and RabbitMQ support multiple messaging protocols, including JMS, AMQP, MQTT, and WebSocket. Kafka, on the other hand, uses its own custom protocol and does not support other messaging protocols.
- ***Message persistance***: ActiveMQ and RabbitMQ both provide support for message persistence, which means that messages are stored in a durable storage location even if the broker goes down. Kafka also provides support for message persistence, but it is designed to store messages in memory for faster access.
- ***Data processing***: Kafka is designed specifically for real-time data streaming and processing, while ActiveMQ and RabbitMQ are more traditional message brokers. Kafka provides a distributed architecture that allows it to scale horizontally and supports features such as fault-tolerant data replication, data retention, and data compression.
- ***Language support***:ActiveMQ and RabbitMQ are both written in Erlang, while Kafka is written in Java and Scala. ActiveMQ and RabbitMQ provide client libraries for multiple programming languages, including Java, Python, .NET, and Ruby. Kafka also provides client libraries for multiple programming languages, but it is primarily used with Java and Scala.
- ***Licensing***: ActiveMQ and Kafka are both available under the Apache License, while RabbitMQ is available under the Mozilla Public License. This may be a consideration for organizations that have specific licensing requirements.

3 Description of the architectures and implementations

All the architectures provide services to the queue or topic destination. We choosed to work on the topic target. We containerized all the services in docker containers, in order to avoid the configuration and to have an automated process each time we want to use one of the three technologies.
For all the services, the broker is configured automatically, both in the design of the structure and in the way the messages are spread.
We choosed to use the Poisson's function in order to manage the rate of the messages in such a way that the interarrival time of the messages was not deterministic.

- In the ActiveMQ implementation we used Java and the JMS protocol as messaging service. The broker is containerized in to a Docker container. In the producer and consumer code we first create a *connection factory*, that is useful to create a connection to the message broker. We create the connection factory using the ActiveMQConnectionFactory class, more precisely the createConnection() method. Once we have created the connection, we created the *session object*, useful to create and send messages. We create the session using the createSession() method of the Connection object. Then we create a destination (topic) thanks to createTopic() method of the session object. In the producer side we create a producer thanks to the createProducer() method of the session object, instead for the consumer side we create a consumer through the createConsumer() method of the session object. The producer can send the message using the send() method, instead the consumer receives the message thanks to a message listener. Once the exchanging of the messages is done, the producer closes the connection, instead the consumer keep listening until you stop it. In this technology the throughput is calculated by the consumer, without making distinctions on who send the message. The message filtering, the transactional messaging and the message persistence are the default ones.

- For the implementation of the Kafka service, we used the Kafka library for python. In the producer we instantiate a new producer object, which connects to a broker, and then using a loop we send the messages. Each message is sent on a chosen topic, along with the payload and a timestamp, which we'll use to compute latency. Right after the loop we send an ending message, used to tell the consumer that the sending is over. In the consumer script, we instantiate a consumer object, specifying the broker we want it to listen to. We pass the topic to each consumer, and we assigned to each consumer a different group id, because we wanted the same message to be delivered to all the consumer subscribed to a topic. We have a loop which takes care of handling incoming messages until it does not get an ending message by the producer. Each consumer can handle messages from different producers, even tough on the same topic. The consumer once received the end message from each producer computer the latency and the throughput related to each producer. One of the main architectural challenges was to choose how to handle the delivery of the sent messages. Kafka default behaviour is to send messages in batches, without any end code or something similar. This means that a message sent by produces is not guaranteed to be delivered . This happens because kafka main use case is data streaming. This is the reason why at the end of the producer flow we have to call the flush function, to make sure that all the messages are sent. As said before, differently from other technologies, kafka does not send messages instantaneously but in batches, this means that the latency, computed from when the message is generated, to when the message is received by consumer is not optimized. Using this solution the latency is around 8-10 ms, which is quite high. We tested a different solution, forcing the producer to send the message as soon as it generates it by calling flush at each message send. This choice improves latency, which decreases to 3-4 ms, but the throughput drops to values 10 times low. In the end we choosed to go on with the first solution accepting an higher latency in order to get a better throughput.

- The RabbitMQ implementation has been written in Java, using the binary protocol AMQP:0-9-1O. The message body sent by a producer needs to be sent as a Byte Array, not as a String. The message properties sent for each message are the ProducerID (generated using java.util.UUID.randomUUID()) and the timestamp (computed in nanoseconds coming from System.nanoTime() as a long) are attached to each message sent. Instead of using the standard properties, they are encoded as "customized" message properties, mapped in a generic Map<String, Object>.

  Once the consumer is loaded it starts listening messages on its topic, while keeping track of the different producers which are sending the message, providing a different statistic for each producer.

In rabbitMQ Exchanges take a message and route it into zero or more queues. We used the Topic Exhange, which has an similiar behaviour to the Direct Exchange: "a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key" (https://www.rabbitmq.com/tutorials/tutorial-five-java.html). It is just more flexible and allow to use composite (hierarchical) topics.

The version of the three services during the experiments were:

service	**version of microservices**	**version of broker**
ActiveMQ	'activemq-core', version: '5.7.0'	Apache ActiveMQ 5.17.2
Kafka	'kafka-python', version '1.3.5'	Apache Kafka: 2.8.1
RabbitMQ	'amqp-client':'5.16.0'	rabbitmq:3-management 3.11.11

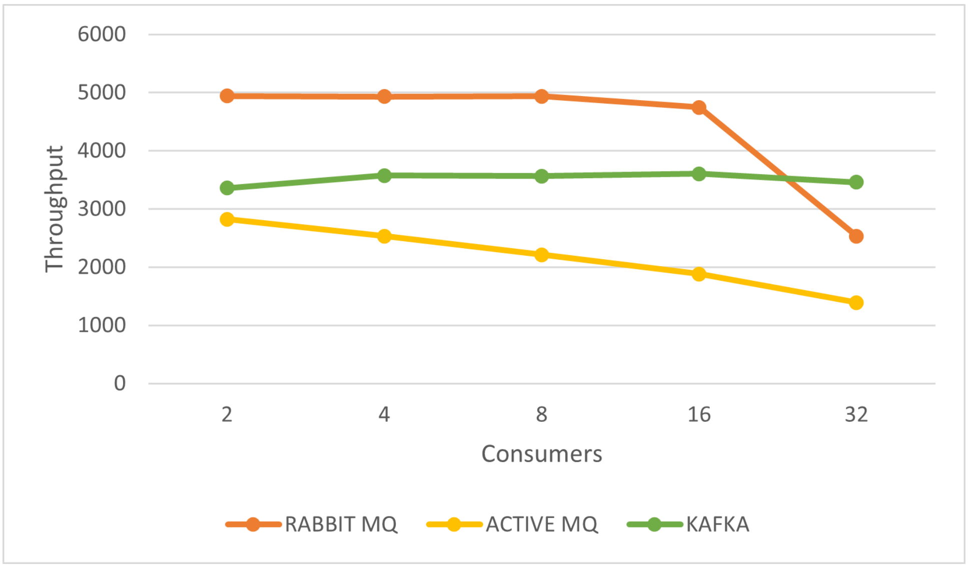
4 Potential limitations

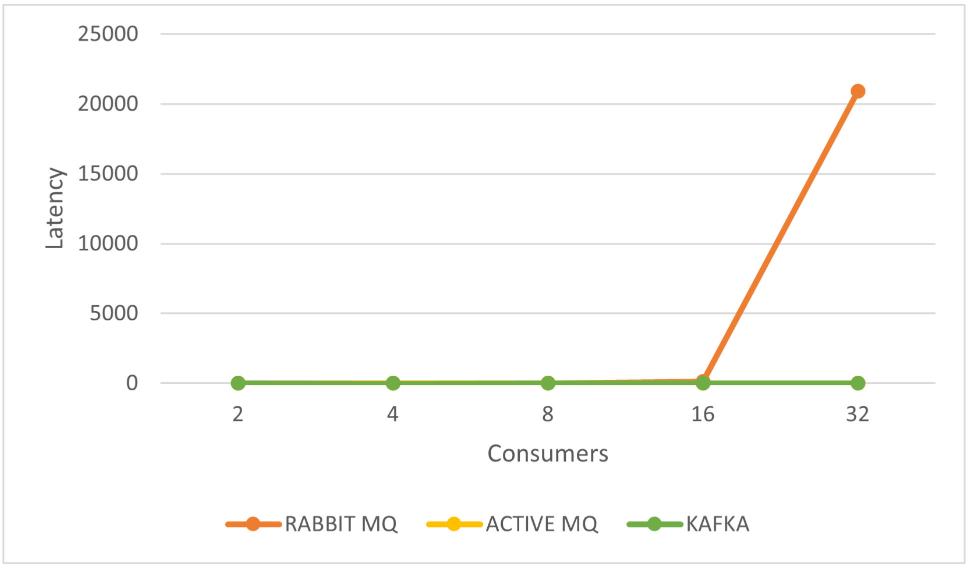
The machine used during the testings, is a Windows 11 Home edition, with an Intel Core i7-10750H CPU (2.60GHz 6 core - 12 logical processor) and 16GB o memory. The version of docker desktop used is the 4.17.0, instead the Docker Engine during the tests was at version v20.10.23.
In the implementation side, we manage the throughput in the three technologies in order to instatiate only one producer for each topic, because it was useless doing the contrary, since we could get the same result simply increasing the number of messages sent by the same producer. The three services are been implemented in two different languages (java and python), so the results of the experiments could be affected by some parameters strictly related to the chosen language.

5 Experiments and evaluations

We decided to compare and to evaluate the performances of the the services under the throughput and latency parameters. It follows the experiments with the related evaluations:

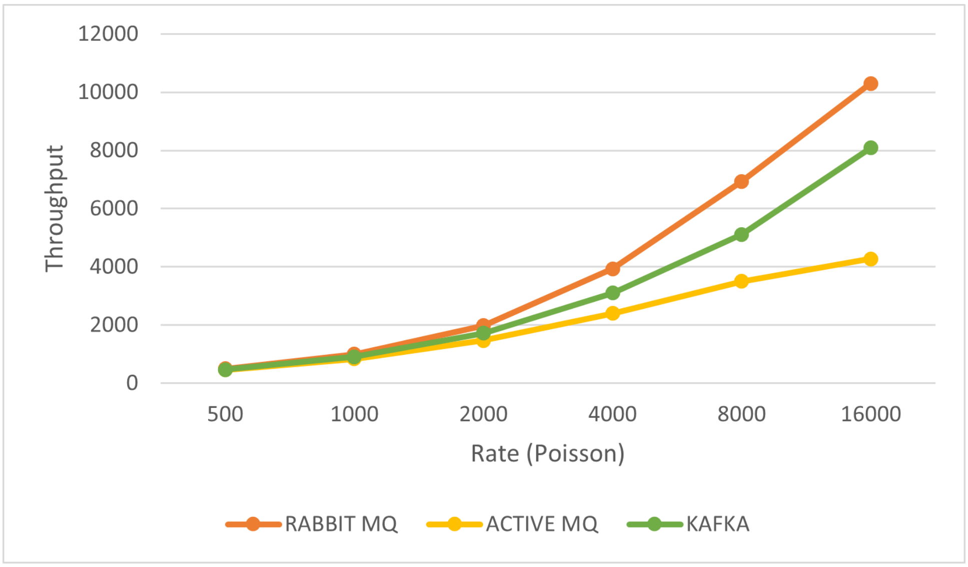
- ***Variable number n of consumer, 1 producer, 300000 messages, lambda=5000, same topic:***

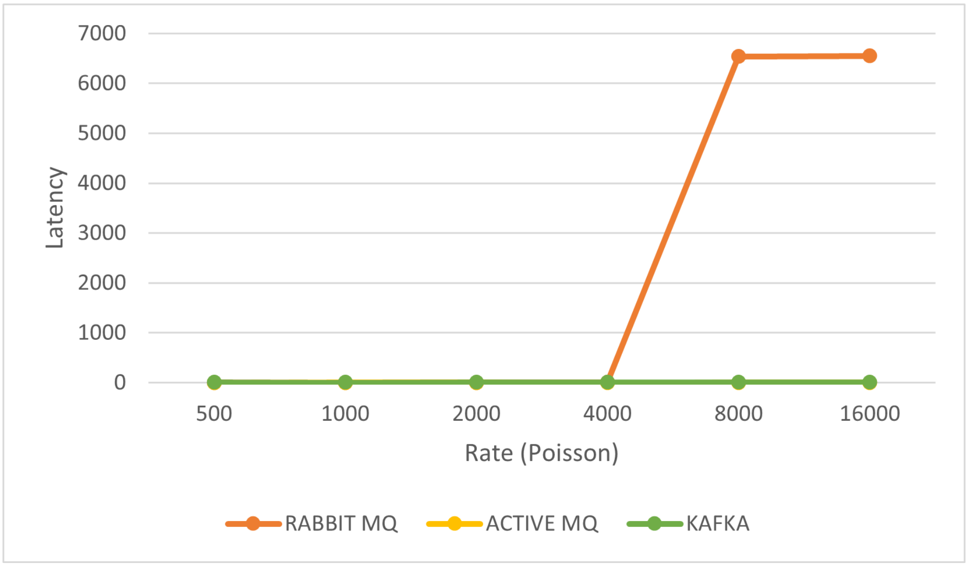




In the first test we instantiated 1 producer and up to 32 consumers, listening messages on the same topics. In terms of throughput RabbitMQ is the service which better performs for a few number of consumers. When the number of consumer grows, starting from 8 the latency starts to increase exponentially making the service inefficient. On the other hand the throughput remain quite stable for all the 3 services.

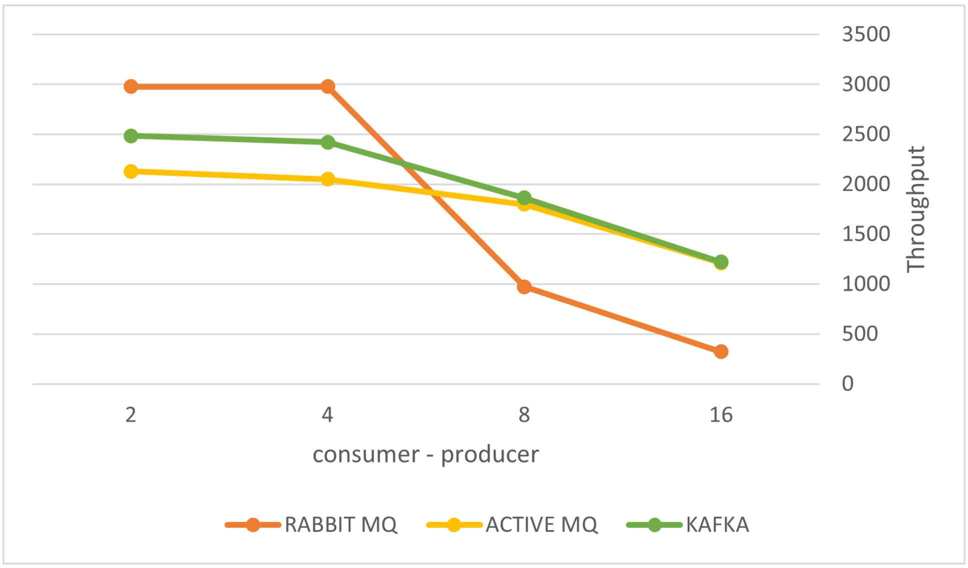
- ***4 consumers, 4 producers, variable number of messages, variable lambda, one topic for each pair producer-consumer:***

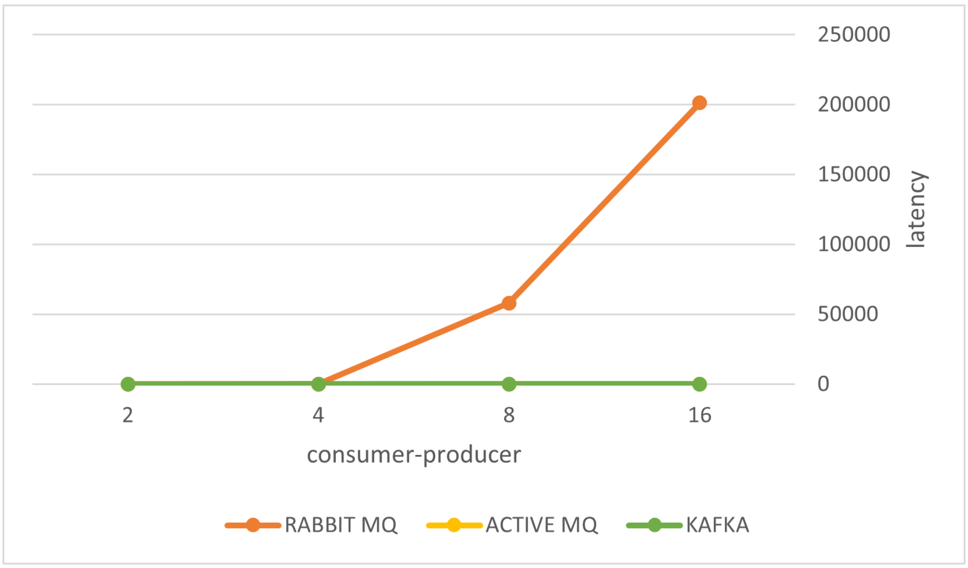




In this test we fixed the number of consumer and producer to 4 and 4, all with different topics and gradually increased the rate of the sent messages. In an ideal scenario the throughput should match the sending rate. Both rates refers to messages sent and delivered on a single topic. We can observe that up to 1000 messages all the services behaves in the same way, but for a large number of messages RabbitMQ takes the lead with the highest throughput. Instead, for what concerns latency, while kafka and ActiveMq keeps latencies of 8ms and 0.2 ms respectively, RabbitMq average latency starts to grow reaching 6553ms with a rate of 16000 msg/s. Analyzing this behaviour we get that in the cases of Kafka and ActiveMq the messages are not 'produced' until the broker is able to deliver them. This allows to keep the latency bounded. Differently for RabbitMQ all the messages are sent to the broker, which can work at the maximum capacity for the whole time, achieving a better throughput at the cost of an extremely higher latency.

- ***variable number n of consumers, variable number n of producer, 180000 messages, lambda=3000; one topic for each pair producer-consumer:*** In this last test we keep constant the message rate to





3000 msg/s and increase the number of the pairs producer/consumer. Each pair work on a different topic and the obtained values refers to a single pair. This is an extremely stressful condition for the broker. As we see in the output graph, all the 3 services can handle up to 4 pairs, then the performance starts decreasing, in particular for what concerns RabbitMQ, whose throughput with 16 pairs drops to only 500 msg/s. A similar thing happens for the latency, but in the opposite way: RabbitMQ reaches a latency of 200 seconds, which makes the service inadeguate for such kind of task. In this case, even ActiveMQ's and kafka's latency increase to respectively 2,67ms and 46ms.

6 Conclusion

From the three experiments explained in the previous chapter, we noticed that rabbitMQ is the most efficient solution for both throughput and latency when we deal with system whose workload does not exceed the capacity of the system. In such case, the performance of RabbitMQ sulution falls to unacceptable values. Both ActiveMQ and Kafka show better resilence under stressful conditions. For what concerns ActiveMQ we have a really low latecy, at the cost of a low throughput, while Kafka has a significant higher latency but with a better throughput. So in the end, if we need a solution with the best performance, we may choose RabbitMQ as solution, if we have to deal with system that whose workload may exceed system capacity, is better to use Kafka if we want an higher throughuput, ActiveMQ if we need a better latency
The source code of the three technologies, with the installation manual for the software configuration and execution, are available at the link:

https://github.com/plamax1/DDS-Project.git