Comparison of Message Queue Technologies for Highly Available Microservices in IoT

Marko Milosavljević, Milica Matić, Neven Jović, Marija Antić

Abstract— **Internet of Things (IoT) solutions connect large numbers of devices, which generate various data and control messages asynchronously. In the IoT system cloud, these messages need to be queued in order to control the processing load and prevent the overload in cases of traffic bursts. On the other hand, one of the requirements the IoT cloud needs to fulfill is the high availability. Therefore, multiple instances of services accepting and processing the messages generated by the devices are needed. There are various message queue technologies available today, but they all have their limitations. In this paper, we compare the performance of Apache Kafka and RabbitMQ in the scenario of the highly available IoT cloud data processing.**

Index Terms— **message queue; high availability; load balancing; internet of things.**

I. Introduction

In the past decade, the world is witnessing the expansion of Internet of Things (IoT) solutions. Within IoT systems, different devices are connected to perform a certain function together. IoT use-cases are various, such as smart transport, smart fabrics, smart cities, smart homes, etc.

In order to collaborate, the devices need to be able to exchange data such as commands and state change reports. Although the expansion of IoT has led to the development of technologies such as ZigBee, Z-Wave, WiFi or Bluetooth Low Energy, which enabled the connection of many different actuators and sensors into large local mesh networks, in order for an IoT solution to achieve its purpose, the existence of the cloud component is also needed. The cloud allows remote control and monitoring of the local networks, but it can also provide advanced features which require processing of larger quantities of historical system data, or the interaction with components responsible for customer management, software update and third-party services.

As the data from the IoT system is generated asynchronously [1], and processing it requires a certain amount of time, mechanisms are needed to control the cloud load. Usually, this control is achieved by deploying various message queueing systems, that allow to communicate between different components of the cloud, and react to messages generated by the end devices [2]. Message queuing technologies which are available today differ in terms of the performance guarantees they offer, and depending on the actual use-case, metrics such as latency, disk space, RAM memory or processor usage may be a limiting factor [2], [3]. The comparison of Kafka and Apache Pulsar has been performed by the authors in [4], and it has been shown that, although Apache Pulsar may achieve better results in terms of resource usage, the maturity of the solution, available documentation, and possibility to integrate with other data processing tools, may be a reason to favor Kafka in the commercial deployment scenarios. On the other hand, Kafka and RabbitMQ have been compared in [5], to show that RabbitMQ has its advantages in terms of the achieved throughput on a single server instance, but the scaling options are on Kafka's side.

In this paper, we explore the possibility of replacing the already implemented RabbitMQ message queueing within the smart home system cloud [6],[7], with Apache Kafka. Within the deployed smart home cloud, messages generated by end devices are processed by multiple cloud services. As the number of supported features is growing, so is the number of the cloud services that process these messages. Also, some of the messages need to be processed by multiple of these services. Additionally, as the number of users grows, the system needs to be scaled up, and, as already said, Kafka has its advantages in this domain. The paper is organized as follows: in Section II, the elements of smart home system and its cloud architecture are introduced, then the overview of RabbitMQ and Kafka is given in Section III and Section IV. Finally, the performed tests and their results are presented in Section V.

II. Smart Home Cloud Data Buffering

In the existing smart home solution, the end devices within the household use technologies such as ZigBee, Z-Wave and ONVIF/IP to connect to the home gateway – Fig. 1. The gateway is responsible to execute the core system logic: it implements the middleware which represents all of the devices in the same way, regardless of the communication technology they use in the local network, and allows them to work together, according to the automation rules set up by the user. To communicate with the user applications and cloud backend, the gateway uses MQTT protocol. MQTT conveys commands issued by the user, system control messages, and reports about device state changes. Control messages are processed on the cloud side, for the purpose of system

Marko Milosavljević is with OBLO Living, Novi Sad, Narodnog fronta 21a, Serbia (e-mail: marko.a.milosavljevic@ oblolivingcom).
Milica Matić is with the Faculty of Technical Sciences, University of Novi Sad, Serbia (e-mail: milica.matic@rt-rk.uns.ac.rs).
Neven Jović is with OBLO Living, Novi Sad, Narodnog fronta 21a, Serbia (e-mail: neven.jovic@ obloliving.com).
Marija Antić is with the Faculty of Technical Sciences, University of Novi Sad, Serbia (e-mail: marija.antic@rt-rk.uns.ac.rs).

administration, upgrade, backup and restore. Also, reports about device state changes are stored to provide user with the information about the history of system usage [7].

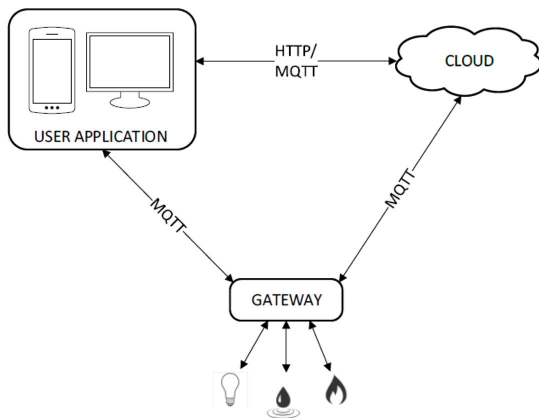

Fig. 1. Smart home system components and communication between them.

The observed smart home cloud system solution has the microservice-based architecture. It is highly available (HA), which means that the entire system is fault tolerant, i.e. that there are multiple instances of every microservice running [6]. In order to prevent problems with MQTT messages processing due to the overload of cloud system, or the failure of some instances, temporary data buffering is necessary. In the temporary data buffering module all important messages are first queued, allowing relevant microservices to process them at their own pace.

In the current implementation, RabbitMQ is used for the purpose of data buffering. The incoming MQTT messages are parsed by the B2Q (Broker to Queue) microservice, and directed to the appropriate RabbitMQ queues, based on the information they contain. All of the instances of one cloud microservice share the load of processing the messages from the RabbitMQ queue they are associated with. The problem here represents the fact that if one message needs to be processed in multiple ways (i.e. it is relevant as the input for multiple cloud microservices), it has to be replicated to multiple queues. Therefore, in this paper we explore the possibility of replacing RabbitMQ with Apache Kafka. We implement the B2K (Broker to Kafka) microservice, which publishes messages to Kafka queues, that the processing microservices are subscribed to, and we compare the performance of the two implementations.

A. RabbitMQ

RabbitMQ is a message queue manager, which has originally implemented the Advanced Message Queuing Protocol (AMQP). Later it was extended to support Streaming Text Oriented Messaging Protocol (STOMP), Message Queue Telemetry Transport (MQTT), and other protocols, but AMQP remains the default and the most widely used one.

RabbitMQ messages can convey any kind of information, from a simple text message to a message with information about processes important for the system. Message broker stores the message into the queue, until the application fetches it for processing. Message queuing allows web servers to avoid the overload, as they can control the number of the messages that are processed simultaneously. It is also useful for distributing messages to multiple consumers sharing the load and providing fault tolerance.

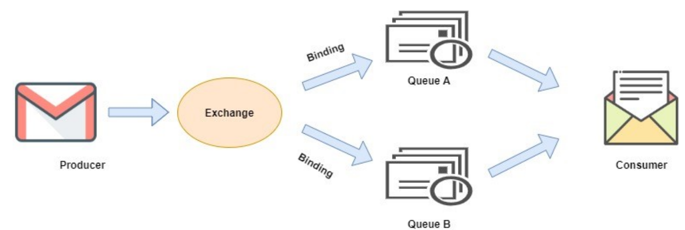

Fig. 2. RabbitMQ message delivery mechanism.

Producer applications create the messages, but the messages are not published directly to a queue. First, the producer sends the message to the RabbitMQ exchange running on the broker – Fig. 2. The exchange is responsible for routing the messages to different queues, based on the configured bindings and routing keys. Four types of exchanges exist - direct, topic, fanout and headers exchange. In the direct exchange, the message is routed to the queue whose binding key matches the routing key of the message. The topic exchange does a wildcard match between the routing key and the routing pattern specified in the binding. The fanout exchange routes messages to all of the queues bound to it. The headers exchange uses the message header attributes for routing. Consumers subscribe to the queues and process the messages from them. All consumers subscribed to the same queue will share the load of processing the messages from that queue. The messages are deleted from the queue after processing.

B. Apache Kafka

Apache Kafka is an event streaming platform. It is elastic, distributed, highly scalable and fault-tolerant. Similar to RabbitMQ, Kafka has the client and server side. Kafka clients and servers communicate using TCP protocol.

Kafka implements the publish/subscribe mechanism, and allows processing streams of events as they arrive into the system or retrospectively, but also allow to store streams of events as long as they are needed.

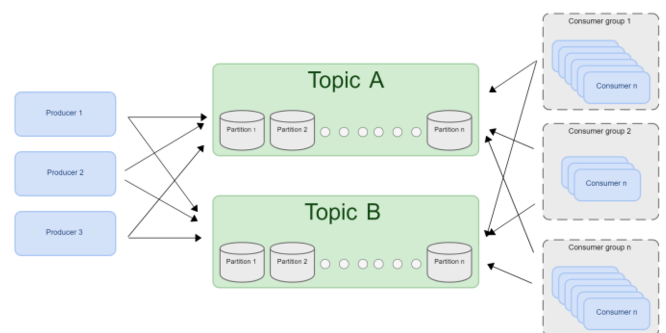

Fig. 3. Kafka message processing mechanism.

Similar to RabbitMQ, the Apache Kafka clients can act as producers and consumers – Fig. 3. Producers represent client applications that write (publish) events to Kafka. On the other hand, consumers are subscribing to topics, reading and writing events. Producers and consumers are not aware of each other. They work completely independently, and that is a key design to achieve high scalability. Therefore, producers will never need to wait for consumers.

When data is written to Kafka, it is written in the form of an event containing the key, value, timestamp and optional metadata. Events are stored in topics. The durability of events inside Kafka's topic is configurable. Unlike RabbitMQ, Kafka events can be read whenever they are needed, because events are not deleted after consumptions. Events can be stored as long as needed. Storing data for a long time does not affect Kafka.

Topics in Kafka are partitioned, and one Kafka topic can have any number of partitions defined in the Kafka configuration file. Events are ordered inside the partition in the exactly same order as they were written, and one consumer can process data from one partition only. However, the data stored in one partition can be processed by multiple consumers belonging to different consumer groups, i.e. one message can be processed multiple times, without the need to duplicate it. Offset is an integer number that is used to maintain the current position of a consumer inside partition. Every topic can be replicated, so that there are dozens of brokers that have a copy of data. This makes data fault-tolerant and highly-available.

III. Testing and Results

Tests were designed to measure CPU load of smart home system servers when RabbitMQ and Apache Kafka are used for data buffering. RabbitMQ and Kafka brokers were run on the 8-core Intel i7 processor with 8 GB of RAM memory.

TABLE I
RabbitMQ test results

Setup	CPU usage on 8 cores [%]		
	average	maximum	deviation
16 producers 8 queues 0 consumers	324	640	108
16 producers 8 queues 8 consumers	410	794	197
16 producers 8 queues 16 consumers	486	800	167
16 producers 8 queues 32 consumers	553	800	147

To test the RabbitMQ buffering, 16 producer B2Q processes were created, that published messages to 8 queues. The messages from these queues were processed by a variable number of consumers (0, 8, 16, 32). Producers were configured to publish messages every 1 ms. Test results are presented in Table I.

RabbitMQ reached CPU limit after 16 consumers, but was able to continue working stably, while the setup with 32 consumers stopped working after ten minutes. The throughput of the system was approximately 11000 messages per second. Maximum CPU usage was 800%, i.e. all eight cores were used 100%.

To test Kafka performance, 16 producers were created, which published to the variable number of partitions (32, 64, 128). Since Kafka allows only one consumer per partition, the number of consumers was also varied from 0 to 128. Test results are presented in Table II.

In any of test cases limit of Kafka maximum CPU load was not reached. It can be observed that the CPU usage deviation is smaller than in RabbitMQ case. Therefore, the server stays stable, even as the number of messages that are stored in Kafka increases with time.

TABLE II
Kafka test results

Setup	CPU usage on 8 cores [%]		
	average	maximum	deviation
16 producers 32 partitions 0 consumers	210	573	65
16 producers 32 partitions 32 onsumers	202	347	43
16 producers 64 partitions 0 consumers	186	473	90
16 producers 64 partitions 64 consumers	208	360	37
16 producers 128 partitions 0 consumers	150	300	93
16 producers 128 partitions 128 consumers	480	553	53

IV. Conclusion

This paper gave a brief description of some of the message queueing technologies that can be used for flow control and load balancing in the IoT scenario. RabbitMQ and Apache Kafka were deployed within the smart home system cloud, and their performance was tested for a variable number of consumers.

The presented test results indicate that data buffering in Kafka is highly stable and has the lower average CPU usage. At any point of testing, maximum CPU usage was never reached. Therefore, in our further work we will focus on integrating Kafka in the data collection and storage module of the smart home system. Using Kafka will allow us to process the same messages multiple times, without the need to duplicate data. This, in turn, opens the possibility to create advanced data processing scenarios which may bring added value to the users of the smart home system.

V. ACKNOWLEDGMENT

This research (paper) has been supported by the Ministry of Education, Science and Technological Development through the project no. 451-03-68/2020-14/200156: “Innovative scientific and artistic research from the FTS (activity) domain”.

REFERENCES

[1] F. Metzger, T. Hoßfeld, A. Bauer, S. Kounev and P. E. Heegaard, “Modeling of Aggregated IoT Traffic and Its Application to an IoT Cloud,” *Proceedings of the IEEE*, vol. 107, no. 4, pp. 679-694, April 2019

[2] G. Fu, Y. Zhang and G. Yu, “A Fair Comparison of Message Queuing Systems,” *IEEE Access*, vol. 9, pp. 421-432, Jan. 2021

[3] H. Wu, Z. Shang and K. Wolter, “Performance Prediction for the Apache Kafka Messaging System,” *Proc. of IEEE HPCC/SmartCity/DSS*, Aug. 2019

[4] S. Intorruk and T. Numnonda, “A Comparative Study on Performance and Resource Utilization of Real-time Distributed Messaging Systems for Big Data,” *Proc. of IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, July 2019

[5] P. Dobbelaere and K. S. Esmaili, “Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper,” *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17)*, June 2017

[6] M. Matić, E. Nan, M. Antić, S. Ivanović and R. Pavlović, “Model-Based Load Testing in the IoT System,” *Proc. of International Conference on Consumer Electronics (ICCE-Berlin)*, Sept. 2019

[7] S. Ivanović, M. Antić, I. Papp, N. Jović, “Data Acquisition, Collection and Storage in Smart Home Solutions,” *Proc. of 6th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN)*, May 2019