Publish-Subscribe Approaches for the IoT and the Cloud: Functional and Performance Evaluation of Open-Source Systems

Apostolos Lazidis¹, Konstantinos Tsakos², Euripides G.M. Petrakis³,*

¹School of Electrical and Computer Engineering, Technical University of Crete (TUC), Chania, Crete, Greece

Abstract

Publish-Subscribe systems facilitate the communication between services or applications. A typical system comprises the publisher, the subscriber, and the broker but, may also feature message queues, databases, clusters, or federations of brokers, apply message delivery policies, communication protocols, security services, and a streaming API. Not all these features are supported by all systems or, others may be optional. As a result, there is no common ground for the comparison of Publish-Subscribe systems. This paper presents a critical survey and taxonomy of Publish-Subscribe systems, of their design features and technologies. The concepts of message queuing, publish-subscribe systems, and publish-subscribe protocols for the cloud and the IoT are discussed and clarified. The respective evaluation is about seven state-of-the-art open-source systems namely, Apache Kafka, RabbitMQ, Orion-LD, Scorpio, Stellio, Pushpin, and Faye. For the sake of fair comparison, a minimum set of common features is identified in all systems. All systems are evaluated and compared in terms of functionality and performance under real-case scenarios.

Keywords: Message Queue; Publish-Subscribe; Benchmarks; Evaluation

*Corresponding author

Email addresses: alazidis@isc.tuc.gr (Apostolos Lazidis), ktsakos@isc.tuc.gr (Konstantinos Tsakos), petrakis@intelligence.tuc.gr (Euripides G.M. Petrakis)
1. Introduction

Publish-subscribe systems handle a wealth of information that is transmitted around the world via Internet. In a real-world case scenario, an academic conference takes place to discuss advances in various fields such as medicine and computer science. In this scenario, each presentation involves a number of speakers that publish information to a group of listeners. The listeners can subscribe to presentations that take place in different locations. This communication pattern is also common in many IoT and Web applications. There can be a central service (i.e. a broker) where applications send information (i.e. presentations) and, this service forwards information to listeners who subscribe to specific topics. Systems that support this type of communication are referred to as Publish-Subscribe [1]. Fig. 1 sketches a basic system with the two publishes and three listeners subscribing to topics A and B.

![Diagram of Publish-Subscribe system]

In Publish-Subscribe communication, the publisher is the service that publishes messages (e.g. events) and the subscriber is the one that receives them. These two services operate independently of each other: the publishers do not need to be aware of the existence of the subscribers, and the subscribers do not need to be aware of the publishers. The broker collects the published messages and forwards them to the subscribers. This decoupling of roles of publishers and subscribers is a feature that differentiates the Publish-Subscribe communication pattern from traditional messaging methods based on the request-response model.

Publish-Subscribe systems must scale-up well to handle large message loads
(i.e. more brokers must be added as needed). A Publish-Subscribe system may feature message queues [2] [3] [4], log files or databases [5] [6] [7] for storing messages. Some implementations feature a replication manager [7] (i.e. to ensure that no message is lost, copies of messages are sent to multiple brokers; if one broker fails, it will be replaced by another), a context manager [5] for context data (i.e. the messages and the subscriptions are Linked Data [8]), a federation service [2] (i.e. data can be moved from one broker to another), security services (e.g. data encryption, user authentication and authorization), a streaming API [7] for filtering messages in real-time, etc. A Publish-Subscribe system defines a message format (e.g. byte array [7], JSON [3] [4], NGISI-LD [5] [6] [9]), and a communication protocol (e.g. TCP [7] or HTTP [5] [6] [9], Webscokets [3] [4], AMQP [2]). The subscribers receive messages from brokers using either a push-based (push policy), or a pull-based approach (pull policy). In a push-based approach, the broker checks the communication and forwards the messages to the subscribers as soon as they arrive. In a pull-based approach, the subscriber polls the broker for new messages. The format (i.e. message structure) most systems choose to exchange messages is JSON, which however, tends to be replaced by JSON-LD [8].

Figure 2: Mandatory (left) and optional (right) services of a Publish-Subscribe system.
There are several implementations of Publish-Subscribe systems that meet
different demands. Most systems ensure reliable and secure communication,
others are designed for faster communication, others for reliability (i.e. ensure
that no message is lost), or for handling large message loads without message
losses. In addition, some features can be optional (i.e. they can be activated by
the user). Fig. 2 shows the sets of mandatory and optional services. Each sys-
tem brings a different set of features onboard an application, and the decision
on which solution is the best fit for a particular use case is not always easy. Each
solution must be evaluated considering the requirements of an application (e.g.
whether security or scalability is more important than error tolerance). Many
popular solutions are open-source, while others are fully proprietary, or have
been incorporated into commercial (cloud) platforms. The latter are known
to be highly scalable, fault-tolerant, secure, can be ported to other platforms
(e.g. stream processing), and, most importantly, all provide good performance
guarantees. In fact, choosing one or another solution is not a real dilemma as
all of them support the most essential features.

The present study is about open-source high-performing systems rather
than commercial ones. Most of them have not been studied before, or have
not been compared against each other or against state-of-the-art systems such
as Apache Kafka [7] and RabbitMQ [2]. The most critical factors of systems
design and functionality and their impact on performance are identified, ana-
yzed, and evaluated. In these systems, the broker has a central role, it does
all the heavy lifting in the communication, and runs on a server (e.g. in the
cloud). For reasons of a fair comparison, a minimum set of common features is
applied to all systems. Performance is the most important factor in the compar-
ison and reveals how fast the communication takes place and how it is affected
by scaling the message load. Most previous studies did not consider more than
two or three systems [10][11]. However, the list of open-source systems is still
long, new systems continue to emerge, and a comprehensive comparison of all
is almost impossible.

Publish-Subscribe methods designed for the IoT are a separate category
and are based on network protocols operating on a network of devices. Network layer protocols with publish-subscribe capabilities are discussed as well. AMQP [12] and MQTT [13] are representative examples of this category of protocols. In the absence of a cloud, content-based [14, 15] and SDN publish-subscribe protocols have a role. They are particularly well suited for the communication of resource constraint devices in an IoT network and the fog. However, their maturity is still lagging and their performance can’t be compared with that of publish-subscribe protocols (with a broker) in the cloud.

Concepts and technologies related to Publish-Subscribe systems, design principles, and functionality are discussed in Sec. 2. Results from previous surveys and system comparisons in the field are discussed. These are discussed in Sec. 3. The evaluation of network and content-based protocols is outside the scope of this work. The focus of this work is on systems operating in the cloud. Sec. 4 presents the seven open-source and state-of-the-art systems which are evaluated. All systems are evaluated in terms of functionality and performance. The most critical factors of their design and functionality and their impact on performance are identified, analyzed, and measured in Sec. 5 and Sec. 6 respectively. Lessons learned and guidelines for selecting a system given the requirements of an application are discussed in Sec. 7.

2. Related Work and Background

The concepts of message queuing, publish-subscribe systems and publish-subscribe protocols are inter-related but, not identical. The following discussion identifies similarities and differences between them.

2.1. Message Queuing Systems

These are services to facilitate machine-to-machine (m2m) communication. The messages are generated at one end (the producer) and are delivered to the other end (the consumer). A message queue stands between communicating services or systems. A message queue is a simple FIFO data structure holding
messages waiting to be processed. The decoupling of the communicating ends is the major advantage of message queues.

A message queuing system [16], apart from a queue, implements additional functionality that varies from one implementation to another. Depending on the use case, a queuing system may be scalable (i.e. multiple queues can work in clusters to deal with large workloads), error-tolerant (i.e. the queues hold message replicas so that no messages are not lost in case a queue fails), persistent storage (i.e. the messages are stored on a database on the disk), asynchronous or synchronous communication (i.e. messages enter the queues any time or after the last message has been read and an acknowledgment has been received). Depending on the implementation, the messages can be sent using different message formats (e.g. text, byte arrays, JSON, or JSON-LD). The messaging system can be enriched with security features (i.e. user authentication and authorization, encryption before transmission and decryption at delivery), notifications (i.e. the producers receive notifications on message receipt by the consumers), filtering (i.e. queries run as messages flow in streams to identify messages satisfying some criteria).

2.2. Publish-Subscribe Systems

Publish-Subscribe systems combine the benefits of message queues such as (a) loose coupling (i.e. the publishers and the subscribers need to be aware of the topics but, not of each other), (b) many to many communication (i.e. many publishers can send messages to many subscribers), (c) fast communication (i.e. messages are consumed faster by many subscribers). Most implementations employ a message broker that implements the queuing system. The broker is responsible for accepting and keeping track of the subscriptions, for relaying new data to the subscribers, and may also implement more functionality on its own accord such as mechanisms for reducing bandwidth utilization. By processing data while on the network, the amount of data can be reduced, thus improving bandwidth utilization and providing the receiver (the subscriber) with refined data.
2.3. **Commercial Publish-Subscribe Systems**

Google Cloud Pub-Sub[^1] is a proprietary solution for the Google Cloud platform. Publisher applications can send messages to a topic, and other applications can subscribe to that topic to receive the messages. Both, push and pull message delivery policies are supported. It is an asynchronous messaging service, it provides persistent message storage and can be used for porting messages to a stream analytics platform. The PubSub Connector pulls messages from data sources or processes to feed Google Pub-Sub.

Amazon Simple Queue Service[^2] (SQS) offers two types of message queues: (a) Standard queues offer maximum throughput, message ordering, and at-least-once delivery, (b) SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the order they are sent. Amazon SQS applies server-side encryption (SSE) to messages. Everyone subscribed to a topic receives all messages on that topic. Amazon Simple Notification Service[^3] (SNS) is used to send individual messages or bulk messages to many subscribing clients (e.g. Amazon SQS). Its main function is to push messages to multiple subscribers as soon they are received from the publishers.

In Amazon SQS, messages can be stored for a short duration of time (i.e. 14 days maximum) but, they are not forwarded (i.e. pushed) to subscribers (i.e. the subscribers must poll and pull messages from SQS). Polling introduces some latency in message delivery in SQS. With Amazon SNS, messages are pushed to subscribers immediately. Amazon SNS supports several endpoints such as email, SMS, HTTP endpoint and, Amazon SQS. Amazon SNS and SQS are services of Amazon AWS[^4].

IBM WebSphere MQ[^5] (IBM MQ) is used to connect applications and services across systems. Systems or services communicate with each other by

[^1]: https://cloud.google.com/pubsub/
[^2]: https://aws.amazon.com/sqs/
[^3]: https://aws.amazon.com/sns/
exchanging messages on queues. Subscribers connect to queues to retrieve messages on topics of interest. IBM MQ can encrypt and sign messages using the Advanced Message Security (AMS) service and, also secure the transmission using SSL/TLS. If a connection is temporarily unavailable, IBM MQ queues messages and forwards them to the subscribers when the connection is re-established.

Confluent Platform enriched Apache Kafka with advanced capabilities that ease application development and connectivity across systems. It provides REST APIs for administration and end-user operations, GUI management and monitoring, logging of user actions in Kafka topics and, enterprise-level users support. Advanced functionality not provided by its open-source version includes options for deployment on servers, Virtual Machines (VMs), or Kubernetes. It supports connectors to databases for offloading older data on disks, dynamic load-balancing (e.g. automatic allocation of partitions across brokers in a cluster), enterprise-level security, and fail-safe brokers federation across systems.

2.4. Evaluation and Benchmarks

Esposito [1] dictates the satisfaction of the properties of System agreement, System validity and, System integrity to guarantee reliability. System agreement suggests that, if a non-defective subscriber receives a message, then all non-defective subscribers will also receive the message. System validity requires that if a non-defective publisher sends a new message, then at least one of the non-defective subscribers will receive it. Finally, system integrity requires that each non-defective subscriber must receive each message at most once. These properties are not sufficient to make a Publish-Subscribe model reliable. Often, real-time applications impose strict control on transmission and require transmitting messages within a time frame, otherwise, the messages may be lost. Some researchers considered timeliness property to satisfy that all subscribers

---

6https://www.confluent.io
receive messages within a time limit [17].

Publish-Subscribe system components often behave in an undesirable way due to various errors including node crashes, churn, network anomalies, and link crashes [18] [19]. These errors need to be addressed to implement a reliable Publish-Subscribe system. Two definitions related to reliability are referred to as resiliency and reconfiguration [20]. The former refers to the ability of a system to guarantee the properties of the agreement, validity, and integrity (in the event of network errors) while the latter refers to the ability of a system to handle errors so as not to affect system connectivity.

Performance is an important factor to consider in Publish-Subscribe systems [21]. However, addressing scalability in these systems is not a trivial task. As the message load scales up, system reliability is being challenged and often performance is being affected. Oh, Kim and Fox [22] address the problem of performance. They propose a cost model for a real-time domain and compare a Publish-Subscribe communication model with request-reply and polling models. Esposito [1] addresses the issue of reliability, that is the capability of the system to tolerate faults in the network or in computing nodes, and to support reliable event notification. Xiao, Zhang, and Chen [23] apply a tree structure of topic clusters that saves memory and improves the real-time performance and reliability of message forwarding. Khoury et al. [24] address the problem of privacy of the communication that typically appears when information about the published content and the subscriber interests are exposed at the broker.

There are no standard benchmarks (i.e. message loads, metrics, and tools) for measuring the performance of Publish-Subscribe systems. The Open Benchmarking Framework (OMB) [25] provides benchmarking suites (i.e. workloads and metrics) tailored to Apache Kafka, Apache RocketMQ [26], Apache Pulsar [27] and, RabbitMQ. For reasons of a fair comparison, both OMB and the systems need to be fine-tuned on the same platform but this is done only for the AlibabaCloud or Amazon Web Services (AWS) cloud platforms.

In practice, users are concerned about performance and which system is faster for the application they are interested in. Throughput and latency are two
common performance metrics of performance. Latency is a tangible metric that measures the time it takes a system to deliver messages to the consumers (end-to-end latency). Throughput measures the number of messages produced or consumed per second and depends on how efficiently a system utilizes the hardware (e.g. the disks, RAM and CPU). The highest throughput for a system requires a high degree of customization of both, system and hardware. This requires that the user be familiar with the peculiarities of both but, this is not easy. On the other hand, Publish-Subscribe systems are not equivalent in terms of parameterization (the same parameterization does not apply to all systems). Compared to all other systems, Apache Kafka and RabbitMQ offer the highest degree of parameterization. For example, the user of Apache Kafka can define many partitions for a topic or how many copies of each message will be sent. RabbitMQ (the same as many other systems) does not support partitions. To match Apache Kafka’s setup, a single direct exchange (equivalent to a topic) can be linked to queues (which must be defined as durable).

Additional studies on the Web compare the performance of specific systems. Yigal [11] compare Apache Kafka [7] with Redis [28] (i.e. a cache database that can be used as Publish-Subscribe system) for message processing in log aggregation applications. Nikhil and Chandar [10] present a benchmarking approach for Apache Kafka, Apache Pulsar [27] and RabbitMQ [2] focusing on system throughput and system latency. The work by Raje [29] evolved in parallel to ours and compares RabbitMQ, ActiveMQ [30], and Apache Kafka in terms of latency (i.e. end-to-end time to deliver a large number of messages) with message size as a parameter. The comparison reveals that Apache Kafka and RabbitMQ outperform ActiveMQ in all cases.

3. Publish-Subscribe Systems for the Internet of Things (IoT)

The loose coupling between publishers and subscribers and the multicast messaging support make the publish-subscribe pattern of communication attractive in IoT use cases. In a typical IoT scenario, the devices connect to a
gateway that runs the sensor interface service of an IoT software platform [31].
The sensor interface service publishes JSON messages to the Publish-Subscribe
service (e.g. Orion-LD) of the IoT platform. Each time a new sensor registers
to the IoT platform, a new entity is created in the Publish-Subscribe service.
When a new sensor value becomes available, this component gets updated,
and a notification is sent to all subscribed entities (i.e. users or services) [32].

In a different use case scenario, the devices communicate with each other
similar to the way peers communicate with each other in a peer-to-peer net-
work [33, 34]. Each node (device) can act as a publisher which distributes
messages to one or many devices subscribed to this information based on a
number of criteria. A bottleneck is the filtering of incoming messages espe-
cially when the number of messages or the number of subscriptions is large
[35]. This type of publish-subscribe communication is often supported by a
communication protocol. Here, the concept of the Publish-Subscribe system
is related (and sometimes confused) with that of the Publish-Subscribe pro-
tocol which is used to support the communication in the network of devices.
Apart from communication protocols that use a broker for message forward-
ing (e.g. MQTT, AMQP), multi-cast protocols distribute the messages directly
to the subscribers via intermediate nodes (e.g. DPS, DDS, XMPP) [36, 37]. By
combining features of multicast protocols and Software-Defined Networking
(SDN) a third category of protocols emerged recently [38].

3.1. Publish-Subscribe Protocols

HTTP (HyperText Trasfer Protocol) is an application layer protocol that is
widely used to support RESTful communication of services in the cloud and
over TCP. Due to its high overhead (i.e. high power consumption, header size
and complexity of handshakes), HTTP is not suitable for the IoT and resource
constraint devices that exchange small amounts of information and are not con-
nected to a sustainable power source. It implements a request-reply commu-
nication where the server responds to the requests of a client. This is good for
communication between services but not for devices that send information to
a server without a prior request. CoAP is a lightweight protocol over UDP. It is similar to HTTP but for resource constraint environments. CoAP and HTTP implement client-server communication and are not Publish-Subscribe protocols themselves.

3.1.1. Publish-Subscribe Network Protocols with a Broker

MQTT (Message Queuing Telemetry Transport) [13, 39] implements a many-to-many asynchronous communication over TCP (other protocols such as Bluetooth have been used as well). It is designed to consume low power and small bandwidth which is essential for resource constraint environments such as the IoT. There are two categories of entities, the broker and the clients (i.e. IoT devices). The devices connect to a broker to publish messages on a topic, or to subscribe to a topic. The brokers run on a server and route the messages they receive to the devices or services subscribed to the topic of each message [40]. There can be several broker servers that exchange data. The messages are deleted as soon as they are read. MQTT guarantees message delivery depending on the appropriate quality of service agreement between the publishers and the subscribers (i.e. each message can be sent at most once no matter whether it is received or not, or can be sent at least once provided that an acknowledgment of receipt has been received by the publisher). MQTT brokers may require a username and password authentication from clients to connect. To ensure privacy, the TCP connection may be encrypted with SSL/TLS.

AMQP (Advanced Message Queuing Protocol) [12] is a message queuing protocol. AMQP is an application layer protocol that allows two processes to communicate across IP networks by exchanging messages. The subscribers and the publishers communicate by sending and requesting messages from message queues. It is mostly used to support Publish-Subscribe communication in the cloud rather than in the IoT. It is designed for message communication between systems and services, and together with HTTP, it is among the most heavily used protocols in the cloud. It competes in the same landscape as MQTT but it is more advanced. Because of its relatively low overhead charac-
characteristics and low power demands, it offers a good solution for the IoT as well. It features a message broker that comprises the queues and the exchanges. The messages are published to the exchanges and their role is to route the messages to the queues. The bindings are rules that control the way exchanges are associated with queues. A newer version of the protocol supports broker-less communication between publishers and subscribers. AMQP is a secure protocol that applies TLS for encryption, and SASL (Simple Authentication and Security Layer) for user authentication.

3.1.2. Content-Based Publish-Subscribe Protocols

DDS (Data Distribution Services) \[14\] is a real-time, asynchronous, data-centric (or content-based) protocol proposed by the Object Management Group (OMG). The publishers and the subscribers can communicate asynchronously as peers in a peer-to-peer network based on interests. It is specifically designed for the IoT and small devices that publish data but can be used with powered devices equally well. OpenDDS\[8\] is the implementation of the Object Management Group (OMG) and shares many common features with the Distributed Publish and Subscribe (DPS) \[15\] protocol of Intel. DPS can be regarded as just another open implementation of DDS. These protocols are used to connect devices (or applications) in a distributed environment over a network. Both, DPS and DDS use UDP by default, but they can support TCP as well. Both, support TLS or DTLS security depending on whether TCP or UDP is used, respectively.

DPS uses hop-by-hop routing to forward messages to subscribers in a network. There is no broker and the devices form a dynamic connected mesh where each node (e.g. a device) operates as a publisher that routes the messages to other nodes (subscribers) based on interests (topics). Any device (network node) can publish topics or subscribe to topics. The nodes can be configured to connect to other nodes, or can use a directory service to locate nodes on a topic of interest (i.e. they maintain routing tables based on subscription

---

7 https://www.omg.org
8 https://opendds.org
topics). The messages will be received by all devices subscribed to the same topic. If there are no subscribers for a topic the publication is terminated.

DDS implements a discovery process based on a built-in discovery protocol that facilitates subscribers to specify what information they are interested in, discover, and connect with publishers offering this information. The publishers connect to subscribers on the same topic and with the same Quality of Interest (QoS) policy. To send a message, a publisher declares a topic and a QoS policy. QoS policies rule when the data is sent which, in turn, controls how and when data are accessed and how the interactions between the nodes are established. A distinctive feature of DPS is that messages flow through the network in all directions and across multiple routes. One-to-one (i.e. unicast) message communication is also possible but messages on a topic are forwarded to nodes subscribed to the same topic. DPS implements a TTL (time-to-live) mechanism on devices (i.e. messages are retained for delivery for a TTL period). This feature enables support for devices that are only periodically active (sleepy nodes). For example, a wireless sensor publishes measurements with a TTL (e.g. 5 minutes) and then switches to low power mode. The most recent measurement will be available to the subscribers when they join the network.

The Real Time Publish Subscribe (RTPS) wire protocol of OMG can also be used to support connectivity across applications using different implementations of DDS. XMPP (Extensible Messaging and Presence Protocol) is a client-server protocol with Publish-Subscribe capabilities based on topics. It establishes a two-way communication between two or more network entities. The messages are encoded in XML and are broadcasted over TCP to all network entities subscribed to the specific topic. The subscriber sends queries to subscribe to a topic on the server. It is good for instance message forwarding (like e-mail) including chat and video calls. Google Hangouts and WhatsApp Messenger are built on XMPP. It is scalable and secure with the aid of SASL and TLS protocols.
3.1.3. Software-Defined Network (SDN) Publish-Subscribe systems

DDS is an elegant content-driven publish-subscribe protocol that works at an abstraction layer of the underlying network (DDS itself is not a network protocol). However, the effectiveness of the underlying network infrastructure has a certain impact on the performance (e.g. speed), scalability, and security of the publish-subscribe solution. Koldehofe et al. [43] and Zhang and Jacobsen [38] observed that, publish-subscribe operations correspond to the topology construction of the underlying network. Their works proposed a reference architecture for the Internet of Things that led to the generation of SDN-based publish-subscribe systems. Their architecture encompassed the idea of a centralized software controller to realize the separation of the control and data plane. The control plane is now contained within the SDN controller which implements the (south-band) interface between the application layer and the IoT devices. The SDN controller relays routing information to the network devices using OpenFlow [44]. OpenFlow is a protocol to program the flow table in the switches and routers of the network. Hakiri [45] proposed a publish-subscribe network solution based on the integration of Software Defined Networking and content-based publish-subscribe protocols such as DDS. Wernecke et al. [46] is just another SDN publish-subscribe strategy using the P4 SDN programming language (it does not rely on OpenFlow).

3.2. Open Issues and Challenges

We surveyed network (application) layer and content-based protocols with publish-subscribe capabilities. AMQP and MQTT are representative examples of the first category. These operate with the aid of a central broker service that runs on a server (e.g. in the cloud). AMQP has been also integrated within cloud-based systems (e.g. RabbitMQ) with excellent results. On the other hand, content-based protocols (e.g. DDS, DPS) are designed to operate in the absence of a central broker and laid the foundation for the implementation of a new generation of SDN-based publish-subscribe protocols.
In the absence of a central entity, content-based (i.e. network-based) publish-subscribe protocols have a role in IoT networks applications. They move overhead from the cloud to the network which, in turn, might slow down the network traffic, especially in a high-throughput environment. Although they have proven to support content-based routing among network components and operate very well under hardware limitations, their performance might be far behind the performance of centralized, cloud-based protocols with a broker. However, the issue of performance requires further investigation, and comparison results (to the best of our knowledge) have not been reported in the literature. In addition, it is not clear how the problem of congestion (e.g. in routing tables for SDN-based systems) is handled in existing implementations of content-based and other network protocols. Challenges are also there regarding security and privacy. These problems need to be analysed further taking into consideration the overhead they bring to the architecture. However, content-based approaches are evolving rapidly, and soon, their technology might reach the level of stability and maturity of their cloud-based counterparts.

In the IoT environment, the role of cloud-based systems is complementary to the role of network-based systems. Cloud-based publish-subscribe systems offer mature and stable solutions for the implementation of real-time, high-throughput systems suitable for the processing of IoT data in the cloud and for the communication with applications or users subscribed to IoT entities. They can handle the wealth of information that is produced in IoT applications, provided that this information has been communicated safely to the cloud. They all operate without any regard for the IoT network and assume that IoT devices are always capable of delivering information to the servers in the cloud. The IoT network can be vast encompassing thousands or even millions of devices serving different use cases and spreading information to the Internet. There is where network or content-based systems have a role. They resume responsibility from transmitting IoT data to and from the cloud with the aid of gateway servers installed at network ends. For example, the gateway can be the host of
the broker server in the case of MQTT and AMQP. This solution moves overhead to the gateway which is in charge of controlling and unifying the world of devices and acts as a cloud proxy. The Web of Things (WoT) Architecture recommendation of W3C [47, 48] sets the requirements for the implementation of such a proxy for interconnecting the brokers with the IoT network and the cloud.

It is likely that none of the surveyed protocols alone is sufficient to cover the entire range of communication needs in the IoT ecosystem that combines both, the transmission of information over the network to the cloud (e.g. for distributing information to end-user or applications for processing) and good performance. There is no unified protocol to cover all these needs in modern application domains such as the industry and the smart cities. This might open the discussion of a new generation of high-level, application layer protocols offering a unified (even higher-level) interface to end-users to manage applications in the IoT, the edge-fog, and the cloud.

4. Publish-Subscribe Systems for the Cloud

The following discussion is about seven popular open-source and state-of-the-art systems namely, Apache Kafka, RabbitMQ, Orion-LD, Scorpio, Stellio, Pushpin, and Faye.

4.1. Apache Kafka

Apache Kafka [7, 49] is an open-source platform for parallel processing of large message loads. Features that make it popular are its customization, portability, scalability, fault tolerance, and security. It applies a custom binary protocol based on TCP and employs a cluster of brokers in order to maintain good performance. The messages are categorized into topics and for each topic, the messages are published to disk partitions. Each publisher specifies the topic and the partitions to publish and writes the messages to the partitions in a round-robin fashion. Each new message is written (as a byte array) at the end
of a partition. If set, an acknowledgment of receipt is sent to the publisher (i.e. that the message was published successfully). This can be either a synchronous or an asynchronous operation. If the operation is set to synchronous, the publisher is blocked until the acknowledgment is received; if the operation is asynchronous more messages can be sent before an acknowledgment is received.

As the message load increases, the brokers, the storage as well as the number of subscribers have to increase too to prevent bottlenecks. If the number of brokers is not sufficient to handle the message load, messages may be lost (the producer requests will time-out). Throughput increases with the number of partitions and with the number of consumers that read messages in parallel. For each read, an offset commit request is generated (i.e. a message is sent back to the broker) and this, in turn, increases the work-load at the brokers. The number of consumers reading messages from a topic, cannot be greater than the number of partitions. Apache Kafka does not feature a database (data are written to files on the disk) but can be connected to one if desired (i.e. via the Connector API that Kafka provides to connect to other systems). With a retention policy in place, messages have a time-to-live. Once expired, the messages are deleted thereby freeing up the disk space. Messages are stored for as long as the user declares (the default is 7 days) or until the topic partitions reach a predefined size limit.

The brokers work in clusters. In each cluster, the broker that started first becomes the controller and controls all others in the cluster. Apache Zookeeper\(^9\) manages the brokers and ensures the fail-safe characteristics of Apache Kafka. If an error occurs and the controller stops working, then Zookeeper is responsible for selecting a new controller. Each broker stores the messages in partitions. The partitions can span multiple copies which are distributed to all broker clusters. If the partitions have copies, one of them is the leader and the rest are followers. The publishers and the subscribers always publish and read mes-

\(^9\)https://zookeeper.apache.org
sages from the leader partition. The followers simply keep message copies. If a leader partition stops working (e.g. the corresponding broker fails), the broker controller will select a new leader.

Apache Zookeeper maintains Apache Kafka metadata (i.e. the partitions of each topic, the broker where each partition is located, how many copies each partition has and, if it is a follower or a leader). The producers connect to the brokers either directly or via Zookeeper (if a producer fails, the messages will not be lost). Zookeeper checks if all brokers are working and is responsible for selecting a new broker controller in case of failure. However, Zookeeper is an extra operation layer in Apache Kafka, and most likely, it will be removed from future implementations.

![Figure 3: Apache Kafka architecture with mandatory and optional services.](image)

Fig. 3 illustrates an Apache Kafka cluster with two brokers (boxes with dotted lines denote optional services). The producers distribute the message load (in a round-robin fashion) to the partitions of each topic (i.e. each topic has two partitions). The cluster comprises the controller broker and its copy (i.e. follower). Each partition has a leader (in either broker) and a copy (i.e. fol-
lower). The producers send messages to both topics which are distributed to two partitions.

The subscribers constantly ask the broker for new messages (i.e. Apache Kafka implements a pull policy). If only one subscriber reads the messages from a topic, and the message production rate is higher than the consumption rate then the subscriber can not respond and read the messages in time. The number of subscribers that read a topic can be scaled according to the requirements of the application. Multiple subscribers can work together in groups (referred to as consumer groups) to read the messages faster. Each subscriber in a group reads messages from a partition. If the number of subscribers is less than the number of partitions, then some subscribers will read messages from more than one partition. Conversely, if the number of subscribers is higher, then some subscribers will not read messages from any partition (they will be inactive). Finally, many consumer groups can read the same topic at the same time. The subscribers read messages from partitions in the order they were sent. Each message has a unique key (offset) that indicates its position in the partition. Thanks to that offset, the subscribers know the position of the last message read. So, if they disconnect from Apache Kafka and then reconnect (e.g. if a broker fails), they can continue to read messages from the position last read. Also, thanks to the offset, a subscriber can start reading messages from any position.

Apache Kafka can enable data transfers across individual clusters of brokers located in different data centers, or geographic regions. This way, an Apache Kafka eco-system comprises a federation of multiple interconnected clusters of brokers. The MirrorMaker mechanism enables data in one cluster to be transmitted or copied to more clusters (e.g. located in different regions). This allows data to be recovered if one cluster fails or, data can be sent to another cluster of brokers for processing.
4.2. RabbitMQ

RabbitMQ \cite{rabbitmq} features several routing mechanisms and message communication protocols. Originally, it was developed for AMQP protocol but can support more TCP protocols (e.g. STOMP and MQTT) to define a set of conventions that rule the communication between the publishers and the subscribers. When a subscriber (or a publisher) connects to a broker, a new TCP connection is initiated (each user has only one TCP connection). If a user needs more connections, these are created as virtual connections sharing the same TCP connection called \textit{channels}. The publishers open the connection and create the channels with the broker. Messages flow through a channel and from there to one or more queues through the exchanges. The producers send messages to the exchanges and not directly to the queues\footnote{https://www.rabbitmq.com/tutorials/tutorial-three-python.html}.

Similar to Apache Kafka, an acknowledgment of receipt is sent to the publisher either synchronously or asynchronously. To distribute the workload and ensure high availability, the user can configure a cluster of brokers or, a federation of brokers, in which case data can move from a cluster of brokers to other clusters. The messages are in JSON format. Messages are stored in queues until they are read by the subscribers. As soon as the messages are read, they are deleted from the queue. Each queue can be declared as \textit{durable}, \textit{transient}, \textit{exclusive}, or \textit{auto-deleted}. If a queue is declared as durable then, if the broker fails, it is saved on the disk and can be re-used. Replicated queues must be durable. If it is transient then, if the broker fails, all messages are lost. If a queue is set to be exclusive, it will be used only by the TCP connection for which it has been declared. If declared auto-deleted, the queue is deleted if there are no more subscribers (i.e. the last subscriber unsubscribes from the queue). The communication is protected by Transport Layer Security (TLS).

Queues can be given additional arguments such as (a) queue (or message) time to live: once this time lapses, the queue (or message) is lost and, (b) the maximum size in messages (or bytes) that a queue can hold; once the queue ex-
ceeds this limit, all messages beyond the maximum size are lost. For each message, the publisher declares its routing key and the exchange to use for routing. Before sending it, each message must be declared either as persistent or as transient. Persistent messages are stored on the disk in case of failures (e.g. a broker fails) provided that their queue has been declared durable. Messages are serialized and form a byte array before they are sent. The messages are sent to the publishers in the order they are sent unless they have been given priority (in which case they arrive before all others). Each queue may have many subscribers. This way, messages in the queue will be read faster (in a round-robin fashion). A subscriber can be declared exclusive to become the only subscriber of a queue. The subscribers can issue an acknowledgment of receipt as soon as they receive a message. If the broker does not receive the acknowledgment (within a certain period of time), the message is either re-routed to be re-sent or, it is deleted. The subscriber may choose a policy and how many times failed messages will be re-routed. Fig. 4 shows RabbitMQ with an AMQP broker.

![RabbitMQ basic architecture.](image)

The **Exchanges** in AMQP protocol are agents that receive and route messages to the appropriate queues. Similar to queues, the exchanges can be durable, transient, exclusive or auto-deleted. They apply a set of rules (called **bindings**) to connect and route messages to queues based on a **routing key**. The exchanges can be declared as (a) **direct exchange**: the messages are sent to queues if their
routing key is the same as the message key, (b) default: every new queue connects with the same routing key, (c) fanout: sends messages to all queues connected with it regardless of the routing key, (d) topic exchange: routes messages to one or more queues if the routing key has wildcards (i.e. the routing key partially matches the queue) and, (e) headers exchange: routes messages to queues based on header names instead of routing keys.

RabbitMQ supports authentication, authorization, and encryption mechanisms. The RabbitMQ administrator can manage system users and their permissions using a Command Line Interface (CLI). RabbitMQ supports LDAP, SASL and OAuth2.0 user authentication and authorization mechanisms and applies TLS to secure its communications.

4.3. FIWARE Publish-Subscribe Systems: Orion-LD, Scorpio, Stellio

FIWARE[12] is an open-source initiative and software platform that supports smart context data management in several application domains (e.g. smart cities, smart energy) based on EU standards[13]. In 2019, the European Telecommunications Standards Institute (ETSI) proposed NGSI-LD[14], a new data exchange protocol based on Linked Data (LD)[14]. NGSI-LD provides an API for publishing, querying, and subscribing to context information. Entities are the key components of NGSI-LD. An entity can be the description of a concept or object, a subscription to another entity, or a property of an entity. An NGSI-LD document is a valid JSON-LD document (but not the other way around). Listing 1 is an example of the NGSI-LD declaration of a room entity with a temperature value (the temperature is a property in the room entity)[15]. The @context label defines a hyperlink that points to the ontology that describes the entity. There are no serialization and de-serialization libraries for NGSI-LD. This is a disadvantage for application developers who must be aware of NGSI-

---

[1]https://oauth.net/2/
[12]https://www.fiware.org
[14]https://www.w3.org/standards/semanticweb/data

23

Listing 1: NGSI-LD model example

```
1  {
2    "id": "urn:ngsi-ld:Room:01",
3    "type": "Room",
4    "temperature": {
5      "type": "Property",
6      "value": 17,
7      "observedBy": {
8        "type": "Relationship",
9        "object": "urn:ngsi-ld:Sensor:01"
10      }
11    },
12    "@context": ["https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
13  }
```

ORION-LD provides a Restful HTTP interface for NGSI-LD context information management. Fig. 5 sketches the basic Orion-LD architecture with mandatory and optional components (shown as boxes with dotted lines). NGSI-LD entities (i.e. messages, subscriptions) are stored in a MongoDB database. A publisher may create context entities and a subscriber may issue a request to search for an entity and subscribe to it or to attributes of that entity. Orion-LD implements a custom push or pull mechanism based on MongoDB. The subscriber will be automatically notified of possible updates of the entities of interest (push policy). Orion-LD awaits confirmation that messages sent by the publisher have been received successfully. On error, the messages have to be re-routed.

The user is opted to configure a federation of brokers with which, one broker can forward queries and updates to another broker. There are two models of federation: the push federation and the pull federation. In the latter, the intermediate brokers do not store any data; they simply forward data to the final destination brokers where it is stored. In a federation setting, a context source is realized as a broker with an NGSI-LD interface that connects to another NGSI-LD broker to share information. In the case of pull federation, the data will be stored in each intermediary broker.
Orion-LD is not tolerant to errors (unless in high-availability mode): if the broker stops working, messages not stored in MongoDB are lost and the communication with the broker is interrupted. In high availability mode, Orion-LD can operate a cluster of brokers\(^\text{15}\). The user must install a scheduler (i.e. load balancer), a set of Orion-LD brokers, and a set of MongoDB databases. One of them will operate as the master and the others are copies of it. To ensure high availability, there can be more than one scheduler. One of them is the master and the others are standing by and ready to take its place if it fails. The scheduler distributes (in a round-robin fashion) the messages to the brokers and from there to the MongoDB databases. If a broker fails (i.e. it stops working) another will take its place. If the master MongoDB database fails, it will be replaced by one of its copies (so that the stored data is not lost). Fig.\(^6\) illustrates the three-layer architecture of a cluster of brokers with scheduler (load balancer) and MongoDB.

Orion-LD implements SSL for encrypting data before transmission, and a *Cross-Origin Resource Sharing* (CORS) \(^\text{52}\) mechanism that controls which addresses are authorized to access a connection and its resources. CORS is an HTTP-header-based mechanism that allows a server to declare who (e.g. domain, port) has access to its resources.

Stellio and Scorpio were developed in parallel to enrich Apache Kafka with REST NGSI-LD interfaces. The two systems are very similar. They differ from Apache Kafka in message format. Similar to Orion-LD, they expose REST APIs for handling NGSI-LD entities (i.e. create, read, update and delete operations). Each system features a database for storing NGSI-LD information. Scorpio connects to a PostgreSQL database, while Stellio connects to a Neo4j database \[53\] for messages and to TimescaleDB \[54\] and PostGIS \[55\] databases for handling temporal and geospatial entities and subscriptions. The messages are stored in the PostgreSQL or Neo4j database respectively, and also in Apache Kafka disk partitions. As Apache Kafka is a central component in both systems, they take advantage of Kafka’s capabilities of customization, portability, scalability, fault tolerance, security, and live-stream message processing. Thanks to Apache Kafka, messages have multiple copies and will not be lost if a broker fails. Unlike Orion-LD, Scorpio supports OAuth2.0 user authentication and authorization mechanisms. Scorpio applies OAuth2.0 for identity and authentication management security. Stellio resorts to OAuth2.0 user authentication and authorization (the service is provided by Keycloak \[56\]). For the time being, both systems do not support secure communication such as TSL/SSL within their services or with external systems. Fig. \[7\] illustrates the architecture of both systems.
4.4. PushPin and Faye

Pushpin [3] is a reverse proxy server with Publish-Subscribe features. Similar to a reverse proxy (which hides the server from its users), Publish-Subscribe systems hide subscribers from publishers, and so does Pushpin. Some of the benefits of using a reverse proxy server in a Publish-Subscribe scenario are (a) load balancing: the reverse proxy server distributes the load to the brokers it connects, (b) content cache: the reverse proxy forwards the requests it receives to the brokers it connects; the broker sends a response back to the proxy; if it receives the same request, it responds with a cached copy and will not forward the request to the broker, (c) data protection: it applies SSL for data encryption and hides from users (and possible malicious users) the IPs of the brokers.

Pushpin adopts GRIP\textsuperscript{16} (Generic Realtime Intermediary Protocol) for creating queues, registering subscribers, and for defining the communication protocol (e.g. HTTP long-polling or Websockets). However, Pushpin is not tolerant to failures, and storing the messages on the disk is the only way to ensure that no messages will be lost. Pushpin provides features to ensure the correct order of messages, to avoid posting the same message more than once and, system monitoring in general. The publishers send messages (in JSON) to pushpin using HTTP or Websockets. The subscribers are not aware of the

\textsuperscript{16}https://pushpin.org/docs/protocols/grip/
queues to which they are connected (i.e. Pushpin is responsible for assigning queues to subscribers). Many subscribers can subscribe to a queue to receive the same messages. Pushpin applies a push message delivery policy. Pushpin uses CORS to authorize users to access resources. Users’ identity and access rights are verified by JSON Web Tokens (JWT) in every request. JWTs are not only encrypted but signed as well. When tokens are signed (using public and private key pairs) the recipient of the request can be sure the senders are who they claim to be (i.e. since only the party holding the private key may have signed it).

![Diagram of PushPin and Faye](image)

Faye is a Publish-Subscribe system based on Bayeux protocol. Similar to PushPin, the communication of publishers and subscribers with the brokers is implemented using HTTP Long-polling or WebSockets. The broker comprises several modules namely, adapter, incoming and outgoing extensions, server, engine and storage. A database (on a disk or RAM where needed) stores user IDs (client IDs), user subscriptions to queues, and messages waiting to be routed to the subscribers. The engine is responsible for registering new users, storing subscriptions and, for transient storage of messages and message routing information. The administrator may run the engine on the Redis NoSQL database in the RAM. The server implements Bayeux exchange protocol and supports handshake, login, subscription, and message publishing functionality. It also confirms that the incoming messages comply with

---

17https://jwt.io
the Bayeux format. The adapter is responsible for exposing the HTTP interface of the server and for accepting user connections to the server. Once a message is sent to the broker, the publisher receives an acknowledgment of receipt (otherwise it is re-routed to the broker). The subscribers can register to one or more queues. Faye applies a push message delivery policy. The communication is protected by TLS and (similar to Pushpin) provides Cross-Origin Resource Sharing (CORS) for authorizing access to resources. To guarantee proper user identification and access authorization to resources (e.g. based on O1.0), the user has to write software extensions on her (or his) own. Fig. 8 illustrates in a common diagram the architecture of both systems.

5. Functional Evaluation of Publish-Subscribe Systems for the Cloud

In the following, all systems are compared in terms of their supported features and operations. Table 1 summarizes the features of all systems.

Scorpio and Stellio are designed to add more capabilities to Apache Kafka. However, they are rather new systems and have not been evolved yet to support the full range of customization of Apache Kafka. For a single topic (as in this experiment), all messages are written by default at the same partition and can be read by only one subscriber.

All Fiware systems adopt the NGSI-LD information model for context data. RabbitMQ, Scorpio, Stellio, and Apache Kafka are error-tolerant and this makes them useful in cases where messages should not be lost (e.g. in bank transactions). For applications that process messages in real-time, Apache Kafka, Scorpio, and Stellio are the best solutions. Orion-LD, Pushpin, and Faye are easy to learn although the related documentation is not extensive. In terms of documentation, Apache Kafka and RabbitMQ are the most complete. However, both these systems, because of their extensive functionality and complexity are the most difficult to learn. Stellio and Scorpio are easy to learn. However, if a developer needs to write software extensions for these systems, she (or he) must be familiar with Apache Kafka. Pushpin and Faye are less complete, but
<table>
<thead>
<tr>
<th>Feature</th>
<th>Kafka</th>
<th>Orion-LD</th>
<th>Scorpio</th>
<th>Stellio</th>
<th>RabbitMQ</th>
<th>Pushpin</th>
<th>Faye</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open-Source</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Language</td>
<td>Scala, Java</td>
<td>C, C++</td>
<td>Java</td>
<td>Kotlin</td>
<td>Erlang</td>
<td>C++</td>
<td>Node.js, Ruby</td>
</tr>
<tr>
<td>Learning Difficulty</td>
<td>5/5</td>
<td>2/5</td>
<td>2/5</td>
<td>4/5</td>
<td>1/5</td>
<td>-</td>
<td>1/5</td>
</tr>
<tr>
<td>Message Format</td>
<td>Byte array</td>
<td>NGSI-LD</td>
<td>NGSI-LD</td>
<td>NGSI-LD</td>
<td>JSON</td>
<td>JSON</td>
<td>JSON</td>
</tr>
<tr>
<td>Stream Processing</td>
<td>Stream API</td>
<td>Stream API</td>
<td>Stream API</td>
<td>Stream API</td>
<td>No info</td>
<td>No info</td>
<td>-</td>
</tr>
<tr>
<td>Broker Clustering</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Using Redis</td>
<td>-</td>
</tr>
<tr>
<td>Broker Federation</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Message Queries</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Message Updates</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Message Retention</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Message Replication</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Acknowledgment of Receipt</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Message Delivery</td>
<td>Pull</td>
<td>Push, Pull</td>
<td>Push, Pull</td>
<td>Push, Pull</td>
<td>Push, Pull</td>
<td>Push</td>
<td>Push</td>
</tr>
<tr>
<td>Message Storage</td>
<td>Disk partitions</td>
<td>MongoDB</td>
<td>PostgreSQL</td>
<td>Neo4j, TimesDB, PostGIS</td>
<td>Queue (disk, RAM)</td>
<td>Queue (RAM)</td>
<td>Queue (RAM)</td>
</tr>
<tr>
<td>Connection Encryption</td>
<td>TLS/SSL</td>
<td>SSL</td>
<td>No Info</td>
<td>No Info</td>
<td>TLS/SSL</td>
<td>TLS/SSL</td>
<td>TLS/SSL</td>
</tr>
<tr>
<td>Authentication, Authorization</td>
<td>OAuth2.0, Kerberos, CORS</td>
<td>LDAP, SASL, OAuth2.0</td>
<td>OAuth2.0</td>
<td>LDAP, SASL, OAuth2.0</td>
<td>CORS, JWT</td>
<td>CORS</td>
<td>CORS</td>
</tr>
<tr>
<td>Protocol</td>
<td>Custom TCP</td>
<td>HTTP</td>
<td>HTTP</td>
<td>HTTP</td>
<td>AMQP, STOMP, MQTT, HTTP</td>
<td>HTTP (Long Polling, Streaming), Web-Sockets</td>
<td>HTTP (Long-Polling), Web-Sockets</td>
</tr>
</tbody>
</table>
in terms of learning difficulty, they are the easiest to learn. They are recommended mostly to less demanding applications where performance or fault-tolerance are less mandatory (e.g. chat applications, collaborative text editors).

Apache Kafka implements a custom binary protocol over TCP. Most systems support communication over HTTP protected by TLS/SSL. RabbitMQ supports a variety of protocols (i.e. AMQP, MQTT, STOMP, and HTTP). Communication in Pushpin takes place via HTTP long-polling, HTTP streaming, and Websockets. While communication in Faye takes place via HTTP, HTTP long-polling, and Websockets. Kafka, Scorpio, and Stellio can connect to other external systems thanks to the Connector API of Kafka. Apache Kafka applies a pull-based approach for message delivery (i.e. the subscriber queries the broker for new messages), while Pushpin and Faye adopt the push-based approach (i.e. the messages are forwarded to the subscribers as soon as they are received). All other systems support both approaches. The publishers at Orion-LD, Scorpio, and Stellio can update stored messages and so does Apache Kafka using Log Compaction (i.e. all messages with the same key in the partitions of a topic are updated).

All Fiware systems support broker federation and broker clustering. Faye can also operate a cluster of brokers to achieve high availability (using Redis). Pushpin and Faye do not support message replication. For each topic, the messages are written to the same queue once. Messages in Scorpio, Stellio, Apache Kafka, and RabbitMQ keep multiple copies in partitions and queues respectively. Scorpio and Stellio also have this feature because they use Apache Kafka. RabbitMQ works similarly to Apache Kafka. If a broker fails, then the communication will be continued by another broker. Broker federation is not similar to clustering and refers to the ability of a broker to send messages directly to another broker automatically. This feature is supported by Apache Kafka and all Fiware systems. All systems can sent an acknowledgment of receipt back to the publisher as soon as each message is written at the broker (i.e. partition or queue). For Apache Kafka, Pushpin and Fiware systems, this feature is optional (it is not clear if this feature can be deactivated in Faye).
In Orion-LD, Scorpio, and Stellio the messages are stored on the disk. In Apache Kafka and RabbitMQ the messages are kept for a certain period of time or until the partitions and queues reach a certain size. In Apache Kafka and Fiware systems, the messages can be read more than once. In RabbitMQ, Pushpin, and Faye, once read, the messages are deleted. In Pushpin and Faye, the messages are sent in message queues and, if there is no subscriber to receive them, they are deleted. In regards to security, all systems support at least basic user identification and authorization mechanisms. Scorpio implements OAuth2.0 user identification. Apache Kafka supports basic OAuth2.0 authentication and Kerberos, and authorization using access control lists. RabbitMQ supports OAuth2.0 basic authentication; users are authorized by issuing commands on a Command Line Interface (CLI).

6. Performance Evaluation of Publish-Subscribe Systems for the Cloud

Experiments by Nikhil and Chandar [10] revealed that Apache Kafka (with 100 partitions) delivered the highest throughput (message load) with very low end-to-end latency (i.e. 5msec latency for 200MB/sec throughput). RabbitMQ can achieve even lower latency but for much lower throughput (e.g. 1ms latency for 30MB/sec throughput). All systems were fine-tuned for Alibaba-Cloud or Amazon Web Services (AWS) cloud platforms. These are challenging results and reveal that Apache Kafka might not be the optimal solution for applications that need a messaging system for relatively small message loads [58].

The purpose of the following experiments is not to reproduce the results of previous studies, but rather to show which solution is best for a small-scale application with low throughput (i.e. small message loads). For these experiments, and for a typical system setup, the end-to-end latency (i.e. delay from the time the messages are sent to the time they are received) is a primary con-
cern. The implementations of all systems are downloaded from Docker Hub\(^{18}\) and all systems are evaluated using their pre-configured parameters. No fine-tuning is applied.

### 6.1. Evaluation Testbed

Each publish-subscribe mechanism is deployed on a Docker runtime environment running on an e2-standard-16 Virtual Machine (VM) with 16 vCPUs and 64 GB of Memory on Google Cloud Platform (GCP). To support persistent storage, an 100GB virtual hard disk is mounted to each VM. For each mechanism, the subscribers are deployed on a second VM (i.e. each subscriber is deployed on a docker runtime environment as a docker container).

Orion-LD, Stellio, Scorpio, Faye, and Pushpin are accessible via a REST API over HTTP 1.1. For stressing each system, A JMeter starter kit is deployed on a Google Kubernetes Engine (GKE) cluster\(^{19}\). It comprises multiple services such as a Telegraf operator which deploys daemons on each VM to collect monitoring metrics, an InfuxDB for storing the collected data, and Grafana for data visualization. The tool can submit up to a very large number of HTTP requests (i.e. messages per second) to a publish-subscribe mechanism. The performance stressing scenario is given to Apache JMeter and then passed to the JMeter starter kit. The performance metrics it produces include average and maximum throughput (in messages per second) that a mechanism can achieve, the number of publishers, the error rate (i.e. messages missed), and response time (from where, end-to-end latency is computed).

Apache Kafka and RabbitMQ are accessible via TCP and AMQP respectively. The open-source Kafka Stress test tool\(^{20}\) is deployed on a separate VM for stressing Apache Kafka. It receives in the input the number of messages per second, and reports at the output the throughput that Apache Kafka achieves and the time it takes Apache Kafka to process and deliver the messages at the

---

18\(https://www.docker.com/products/docker-hub\)
19\(https://github.com/Rbillon59/jmeter-k8s-starterkit\)
20\(https://github.com/msfidelis/kafka-stress\)
output.

For stressing RabbitMQ, the Perf-Test\textsuperscript{21} is deployed on yet another VM. It accepts in the input the IP and the port at which RabbitMQ receives traffic and the names and number of the queues where the messages are sent. The number of publishers sending messages simultaneously is also defined. To achieve the highest traffic, the traffic rate parameter is set to unlimited. Throughput and message response time are reported at the output.

6.2. Experimental Results

The purpose of the following experiments is to study the performance of all systems (i.e. end-to-end latency) for the maximum throughput that each system can sustain so that, the error rate is less than 1% (i.e. at least 99% of the messages are received at the output before they are timed-out). The maximum throughput that can be achieved on GCP (no optimization is applies) is determined by the stressing mechanism. As will be shown in the experiments, the error rate for Apache Kafka and RabbitMQ is always 0 even for the maximum throughput that the stressing mechanism can apply.

The stressing mechanism routes messages to the end-point of the publish-subscribe system using multiple threads simulating the effect of multiple publishers sending messages simultaneously. Latency is computed from the time a message is sent to the time it is received. The average latency is computed over all messages sent in the duration of the experiment (i.e. 5 minutes). For each system, the performance is evaluated with two realistic scenarios (a) one broker and one subscriber, and (b) 4 brokers and 4 subscribers. To examine the dependence of the performance on message size, each scenario was run with message size 100Bytes and 1KByte respectively, thus ending up with 4 experiments. To accommodate the possibility of message loss (i.e. in the second scenario), a system should re-route the lost message (i.e. a message for which an acknowledgment has not been received). This is possible for all systems

\textsuperscript{21}https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/#introduction
except Orion-LD, Pushpin, and Faye which do not support message replication. These systems do not support broker clustering and the second scenario cannot be executed. Table 2 summarizes the parameters for each scenario.

Table 2: System parameters for performance evaluation.

<table>
<thead>
<tr>
<th></th>
<th>Scenario A</th>
<th>Scenario B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message size</strong></td>
<td>100Bytes, 1KByte</td>
<td>100Bytes, 1KByte</td>
</tr>
<tr>
<td><strong>Number of topics</strong></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Number of queues/partitions</strong></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td><strong>Number of subscribers</strong></td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Acknowledgment of receipt is set to asynchronous (i.e. the publisher can send more messages before a confirmation is received). For Apache Kafka, Stellio, and Scorpio, each time a message is read, its offset is published asynchronously (i.e. the reader stores the position of the last message read). Apache Kafka has the option to combine small messages that are headed to the same topic partition into a larger group, called a *batch*. Another parameter is *linger.ms* which represents how long the batched messages are held before they are sent to the broker. Each batch is sent when it either reaches its maximum size or when more than the specified milliseconds have passed. For Apache Kafka and for the systems built on top of it (i.e. Stellio and Scorpio), *batch=1MB* and *linger.ms = 10msec*.

Table 3: Performance results with 1 broker, 1 subscriber and 100Bytes message size.

<table>
<thead>
<tr>
<th>System</th>
<th>Throughput (msgs/sec)</th>
<th>Latency (msec)</th>
<th>error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orion-LD</td>
<td>480</td>
<td>4.30</td>
<td>0.10</td>
</tr>
<tr>
<td>Stellio</td>
<td>76</td>
<td>13,380</td>
<td>0.28</td>
</tr>
<tr>
<td>Scorpio</td>
<td>478</td>
<td>18,850</td>
<td>0.03</td>
</tr>
<tr>
<td>Pushpin</td>
<td>1,280</td>
<td>522.35</td>
<td>0.13</td>
</tr>
<tr>
<td>Faye</td>
<td>2,230</td>
<td>3,670</td>
<td>0.44</td>
</tr>
<tr>
<td>Apache Kafka</td>
<td>80,436</td>
<td>11.90</td>
<td>0</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>61,824</td>
<td>1.29</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 3 and Fig. 4 report performance results for publishing all messages to 1 broker. RabbitMQ is the fastest system followed by Faye and Orion-LD. Faye
Table 4: Performance results with 1 broker, 1 subscriber and message 1Kbyte size.

<table>
<thead>
<tr>
<th>System</th>
<th>Throughput (msgs/sec)</th>
<th>Latency (msec)</th>
<th>error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orion-LD</td>
<td>480</td>
<td>11.43</td>
<td>0.02</td>
</tr>
<tr>
<td>Stellio</td>
<td>89</td>
<td>17,040</td>
<td>0.67</td>
</tr>
<tr>
<td>Scorpio</td>
<td>287</td>
<td>33,100</td>
<td>0.34</td>
</tr>
<tr>
<td>Pushpin</td>
<td>2,760</td>
<td>97.11</td>
<td>0.10</td>
</tr>
<tr>
<td>Faye</td>
<td>2,500</td>
<td>1,320</td>
<td>0.07</td>
</tr>
<tr>
<td>Apache Kafka</td>
<td>68,174</td>
<td>14,70</td>
<td>0</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>25,900</td>
<td>0.30</td>
<td>0</td>
</tr>
</tbody>
</table>

does not store messages on the disk. The message queues in Faye are stored in a memory cache while, those of Scorpio, Stellio, and Apache Kafka are stored on partitions. In Scorpio and Stellio, the messages are also stored in PostgreSQL and Neo4j databases respectively. Pushpin is unexpectedly slow. RabbitMQ, the same as Apache Kafka, is very fast but, not the fastest system. In RabbitMQ, before a message is sent to a queue, it is initially sent to an exchange agent, which is responsible for routing the messages to the appropriate queue (i.e. this introduces a delay). In Pushpin and Faye, messages are sent directly to queues. This has a negative impact on performance when the queues are overloaded with messages. However, the high speed of Faye and Orion-LD is achieved for much lower throughput. RabbitMQ and Apache Kafka are the fastest systems for high throughput. The experiment confirms the findings of previous studies [10]. Orion-LD outperforms all other Fiware systems. Orion-LD does not store messages in queues or disk partitions. Messages are saved on a MongoDB database and features connection pool to connect. These links to MongoDB are reusable and Orion-LD does not need to open a new connection upon each request. The size of the connection pool is set to 10 (by default).

Scorpio and Stellio perform about the same since their architecture is very similar. Both systems are slower than Orion-LD. Prior to publishing a message, they search all messages (i.e. one after the other) in a topic to determine whether the message has been published before. Orion-LD does the same, but searching MongoDB is much faster. In Scorpio, the messages are published in
partitions. Checking whether an entity was created before, the messages in the topic are read one by one. In Stellio, a new topic is created for each new entity with the name of the entity. This operation is faster in Stellio than it is in Scorpio.

Table 5: Performance results with 4 brokers, 4 subscribers and 100Bytes message size.

<table>
<thead>
<tr>
<th>System</th>
<th>Throughput (msgs/sec)</th>
<th>Latency (msec)</th>
<th>error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stellio</td>
<td>106</td>
<td>70,680</td>
<td>0.35</td>
</tr>
<tr>
<td>Scorpio</td>
<td>617</td>
<td>45,250</td>
<td>0.77</td>
</tr>
<tr>
<td>Apache Kafka</td>
<td>79,397</td>
<td>12.50</td>
<td>0</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>31,456</td>
<td>9.38</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6: Performance results with 4 brokers, 4 subscribers and 1KByte message size.

<table>
<thead>
<tr>
<th>System</th>
<th>Throughput (msgs/sec)</th>
<th>Latency (msec)</th>
<th>error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stellio</td>
<td>86</td>
<td>17,940</td>
<td>0.08</td>
</tr>
<tr>
<td>Scorpio</td>
<td>315</td>
<td>69,000</td>
<td>0.98</td>
</tr>
<tr>
<td>Apache Kafka</td>
<td>83,522</td>
<td>11.00</td>
<td>0</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>13,164</td>
<td>0.63</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 5 and Fig. 6 report performance results for publishing messages to 4 brokers (i.e. the messages are distributed to 4 brokers in round-robin). The experiment revealed that Stellio and Scorpio are the slowest systems. They are much slower than Apache Kafka and RabbitMQ which achieve 0 error rate even for much higher throughput. For higher throughput, the error rate of all Fiware systems exceeded the 1% threshold. All Fiware systems do not scale up very well. The poor performance of Orion-LD is due to the connection pool. For many simultaneous requests, the connections are busy and many requests wait for a connection to become available. Another reason is that Orion-LD processes only one request at a time (mutual exclusion or mutex policy). For all other systems (except RabbitMQ and Apache Kafka) for throughput higher than 500 messages per second, their mechanisms became overloaded and the messages could not be handled by the broker. Stellio and Scorpio are much slower than Apache Kafka although both systems rely on Kafka. These are
new systems and have not been evolved yet. What makes Apache Kafka fast is that it uses page cache (i.e. disk cache) to write and read messages. When a publisher sends a message to a partition it is kept in the page cache until the system decides to flush it on the disk. The data transfer time from disk to page cache is very fast thanks to zero-copy technique.

7. Conclusions

Nowadays, various systems, such as fire detection and warning systems, rely on fast and immediate responses to detect hazardous events in real-time. In such systems, performance plays a vital role. In that domain, the transmission of messages from the publishers to the subscribers must be as fast as possible. In addition, the system has to be scalable to handle up to a very large number of users and message loads. A scaling parameter will dictate how many users or messages a system can serve without affecting its performance. However, other important factors for selecting a system are ease of use and functionality.

This study presents a comparison of popular open-source Publish-Subscribe systems with two of their state-of-the-art counterparts namely, Apache Kafka and RabbitMQ. Scorpio is the most complete system overall in terms of functionality. However, compared to Apache Kafka, Scorpio (the same as Stellio) lags in terms of performance. For heavy workloads, Apache Kafka and RabbitMQ have proven to be fast and scalable. They operate a cluster of brokers holding copies of all messages while maintaining good performance under stress. They are high error-tolerant, support live-stream processing operations (i.e. can be linked to external systems), and can be supported by user identification and authorization mechanisms. If speed and fault tolerance are the dominant factors for selecting a system Apache Kafka and RabbitMQ are the right choices. If an application needs to interact with others (e.g. databases,

\(^{22}\text{https://developer.ibm.com/articles/j-zero-copy/} \)
stream-processing, etc.) the most suitable systems is Apache Kafka. It provides an API that allows external systems to connect with minimum implementation logic.

Pushpin and Faye offer the least functionality. They are not fault-tolerant, they cannot connect to other external systems, the data in the queues cannot be updated. Faye can be a messaging solution for applications with no particular performance or security characteristics (e.g. chat, collaborative text editors). Scorpio and Stellio have enriched Apache Kafka with REST NGSI-LD interfaces and allow updates and queries on message queues or databases. NGSI-LD format is not supported by RabbitMQ and Apache Kafka. However, they have not been evolved yet to reach the performance levels of Apache Kafka. There is still a long list of systems to study, including ActiveMQ [30], Apache Pulsar [27], ZeroMQ [59] and Redis [28] and many others.

Acknowledgment

We are grateful to Google for the Google Cloud Platform Education Grants program for providing us accounts on Google Cloud Platform.

References


URL https://www.rabbitmq.com

URL https://pushpin.org

URL https://faye.jcoglan.com


   URL  [https://kafka.apache.org/](https://kafka.apache.org/)

   URL  [https://w3c.github.io/json-ld-syntax/](https://w3c.github.io/json-ld-syntax/)


    URL  [https://www.confluent.io/blog/kafka-fastest-messaging-system/](https://www.confluent.io/blog/kafka-fastest-messaging-system/)

    URL  [https://blog.logz.io/kafka-vs-redis/](https://blog.logz.io/kafka-vs-redis/)

    URL  [https://www.amqp.org](https://www.amqp.org)

    URL  [https://mqtt.org](https://mqtt.org)
URL https://www.dds-foundation.org/what-is-dds-3/#

URL https://github.com/intel/dps-for-iot


URL https://ebooks.iospress.nl/volumearticle/22318

URL https://doi.org/10.1145/2933267.2933305

URL https://link.springer.com/chapter/10.1007/978-3-030-75075-6_38

URL https://link.springer.com/chapter/10.1007/11575771_46#citeas


URL http://dx.doi.org/10.34917/16076287

URL https://activemq.apache.org


URL https://doi.org/10.5383/juspn.14.02.001


URL https://dl.acm.org/doi/10.5555/1947725.1947776

URL https://doi.org/10.1145/3292674

URL https://www.nexpcb.com/blog/different-data-protocols-which-one-to-choose


URL https://link.springer.com/chapter/10.1007/978-3-030-15032-7_91

URL https://www.omg.org/spec/DDS1-RTPS/2.3/Betal/PDF

URL https://doi.org/10.1145/2405178.2405181

URL https://cs.gmu.edu/~yhwang1/INFS612/2013_Spring/Projects/


URL https://www.w3.org/TR/wot-architecture/

URL https://www.cloudkarafka.com


URL https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/


URL https://www.timescale.com

URL https://postgis.net

URL https://www.keycloak.org

URL https://link.springer.com/chapter/10.1007/978-3-030-99584-3_10

URL https://zeromq.org