Meeting IoT Platform Requirements with Open Pub/Sub Solutions

Daniel Happ · **Niels Karowski** · **Thomas Menzel** · **Vlado Handziski** · **Adam Wolisz**

Received: June 19, 2015

Abstract The Internet of Things (IoT) will enable a range of applications providing enhanced awareness and control of the physical environment. Current systems typically sense and actuate physical phenomena locally and then rely on a cloud-based publish/subscribe infrastructure for distribution of sensor and control data to end-users and external services. Despite the popularity of pub/sub solutions in this context, it is still unclear which features such a middleware should have to successfully meet the specific requirements of the IoT domain. Questions like how a large number of connected devices that only sporadically send small sensor data messages affect the throughput, and how much additional delay cloud based pub/sub systems typically introduce, that are very important for practitioners, have not been tackled in a systematic way. In this work we address these limitations by analyzing the main features and requirements of several existing academic and commercial IoT platforms and by evaluating which of those features are supported by prominent open pub/sub solutions. We further carry out a performance evaluation in the public cloud using four popular implementations: rabbitMQ (AMQP), mosquitto (MQTT), ejabberd (XMPP), and ZeroMQ. We study the maximum sustainable throughput under a realistic synthetic load and compare the typical end-to-end delay using traces from the TWIST sensor network testbed at TU Berlin. We find that while the core features are similar, the analyzed pub/sub systems differ in their filtering capabilities, semantic guarantees and encoding. Our evaluation indicates that those differences can have a notable impact on throughput and delay of cloud based IoT platforms.

D. Happ · N. Karowski · T. Menzel · V. Handziski · A. Wolisz
Technische Universität Berlin, Telecommunication Networks Group (TKN), Einsteinufer 25, FT 5, 10587 Berlin, Germany
E-mail: {happ, karowski, menzel, handziski, wolisz}@tkn.tu-berlin.de

Keywords IoT · pub/sub · AMQP · MQTT · XMPP · ZeroMQ · performance evaluation

1 Introduction

We are at the brink of a new era of computing driven by rapid augmentation of physical objects around us with computational and wireless communication capabilities. The resulting network of "smart" objects that interact and exchange information without direct human intervention, the so called Internet of Things (IoT), can offer deep real-time awareness and control of the physical environment and serve as foundation for novel applications in a wide range of domains [7].

Based on enablers like server virtualization, fast networking and reliable distributed storage, cloud computing offers important benefits when used as vehicle for realizing core elements of the IoT technology stack. The flexibility, scalability and usage-based cost model enable elastic matching of the growing communication, computation and storage requirements associated with the IoT applications.

Today, several prominent academic and commercial IoT platforms share a cloud centric architecture similar to the one depicted in Figure 1 [15, 26]. Sensor data from devices are sent to a cloud-based service tier. It provides access to the data as well as additional services, e.g. data storage, analysis, or aggregation, to user facing applications.

The current IoT landscape is characterized by a limited interoperability between vertical silos of proprietary IoT sens-

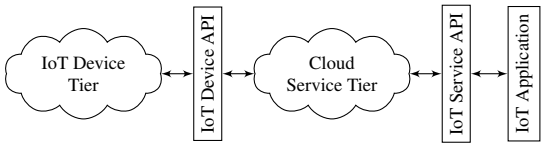


Fig. 1 General architecture of a cloud-centric IoT platform.

Table 1 Survey of IoT platform characteristics

	Architecture			Pattern			Messaging Protocol			Message Format		
	Consumers	Actuators	Gateways	Req/Resp	Pub/Sub	Push	HTTP	Pub/Sub	Other	XML	JSON	Other
Axeda [5]	✓		✓	✓	✓	✓	✓			✓	✓	ASN.1
Digi Device Cloud [10]	✓	✓	✓	✓		✓	✓			✓	✓	
ETSI (one)M2M [30]	✓	✓	✓	✓	✓	✓	✓	MQTT	CoAP	✓	✓	
EVRYTHNG [13]	✓			✓			✓				✓	
FI-WARE [14]	✓	✓	✓	✓	✓	✓	✓			✓	✓	
ioBridge [22]	✓	✓	✓	✓	✓	✓	✓				✓	
openHAB [31]	✓	✓	✓	✓	✓	✓	✓	MQTT		✓	✓	
OpenIoT [32]	✓	✓	✓	✓	✓	✓	✓		Socket	✓	✓	CSV
OSIoT [33]	✓	✓	✓	✓	✓	✓	✓	MQTT/XMPP	CoAP	✓	✓	
RuBAN [9]	✓	✓	✓	✓	✓	✓	✓			✓		
SENSEI [35]	✓	✓	✓	✓		✓	✓		SIP	✓		
Sensor Andrew [38]	✓	✓	✓	✓	✓	✓	✓	XMPP		✓		CSV
Xively [25]				✓	✓	✓	✓	MQTT	Sockets	✓	✓	CSV

ing and actuation platforms. A scalable and open messaging middleware is required to provide connectivity between these silos. Publish/subscribe has become a well established data dissemination pattern for monitoring applications, e.g. stock market or weather data. Due to the similar interaction styles, traffic patterns, and QoS requirements, there is a trend to replace the inappropriate request/response communication pattern in favor of event-driven pub/sub systems in sensor applications [18, 19, 23, 42].

It is still unclear which features pub/sub systems should have to meet the specific IoT requirements, how a large number of connected devices that only sporadically send small sensor data messages affect the throughput, and how much additional delay cloud based pub/sub systems typically introduce. In this paper, we make three main contributions towards addressing the above problems:

- we survey current cloud-centric IoT platforms and derive a set of requirements for pub/sub systems;
- we perform a qualitative analysis of several open pub/sub solutions based on IoT requirements;
- we perform an experimental cloud-based performance evaluation to compare popular open-source implementations of these solutions under typical conditions when deployed in a public cloud environment.

2 Survey of Cloud based IoT Platforms

In this section, we present the result of a survey of several prominent cloud-based IoT solutions. The key findings are listed in Table 1, from which we derive requirements for pub/sub protocols used in IoT platforms.

Architecture – Almost all surveyed platforms explicitly differentiate between data generators and *consumers*, offering separate APIs for interacting with the cloud layer from devices on one side, and from applications on the other. As a result, their architecture is similar to the one depicted in Figure 1. Xively uses a unified API towards the generators and consumers of data. *Actuators* are supported by most solutions. *Gateways* are used to mediate and integrate devices that lack native IP connectivity. We conclude that a pub/sub solutions should support *monitoring and controlling* of sensing and actuation devices. As gateways are present, pub/sub solutions do not need to run on sensor devices directly.

Messaging Pattern – All solutions support a synchronous *request/response* pattern, but most also offer an asynchronous interface, either offering *pub/sub* or *push* based notifications. The push notification service enables applications or users to be notified by the IoT platform when predefined conditions are met. The specification of conditions ranges from notifications on every sensor reading to complex rules including comparison or logical operators. We conclude that IoT platforms should support both synchronous as well as asynchronous messaging. Pub/sub is the dominant pattern for efficient data distribution. The push-based interfaces indicate the need for advanced filtering capabilities in the cloud.

Messaging protocol – Providing uniform access to the data collected is one of the core features of every IoT platform. The results of our survey show that all current platforms support HTTP access. A few platforms offer access via standard pub/sub protocols, notably MQTT and XMPP. We conclude that in contrast to closed off silo solutions, future IoT systems will use standardized protocols with royalty-free specifications and open-source implementations.

Messaging format – *XML* and *JSON* are the leading message encoding formats and many platforms support both. This emphasizes the need for standard formats in IoT settings. Also, both solution offer extensibility to support unforeseen new technologies, which should be offered by future IoT related pub/sub systems.

QoS – While some of the platforms [10, 30, 31, 35] make use of *QoS* capabilities of the underlying network, only Axeda uses message prioritization and OpenIoT supports QoS between the application and device tiers.

In addition to the requirements mentioned above, there are several general non-functional requirements for a cloud based pub/sub system: Due to the large number of devices expected, the system should be scalable. As data may be urgent, messaging should have low latency. Because gateways may be connected over constraint links, protocols and serialization formats should be efficient. Based on these functional and non-functional requirements, we develop a taxonomy of current open pub/sub systems in the following section.

3 Qualitative Analysis of pub/sub Protocols

A publish/subscribe system is a message-oriented middleware (MoM) [6,8,12,39] providing distributed, asynchronous, loosely coupled communication between message producers and message consumers. In the context of IoT platforms we see mainly sensor devices and their gateways in the IoT device tier as message producers, while IoT applications interested in sensor data are regarded as message consumers. An actuator would also act as a consumer.

A pub/sub middleware in general offers three main types of decoupling [12] which make them particularly suitable for large-scale IoT deployments: 1) Message producers and consumers are decoupled in time, i.e. they do not have to be connected at the same time. 2) Messages are not explicitly addressed to a specific consumer but to a symbolic address (channel, topic). 3) Messaging is asynchronous, non-blocking.

A pub/sub system may support different types of filtering, mostly based on topic or content. In the topic based scheme, the symbolic channel addresses are topics, usually in the form of strings, i.e. producers publish to and consumers subscribe to topics. Messages are only delivered to matching subscribers. Topics may be organized hierarchically, i.e. a topic may be a subtopic of another topic. Subscriptions on a parent topic will then usually also match all subtopics. Topic based filtering is a static scheme offering only limited expressiveness. In contrast, in the content based scheme, subscribers are not statically matched based on topics, but dynamically on the content of individual messages, e.g. if a value reaches a certain threshold predefined by the subscriber.

Building on the general requirements for IoT platforms, in this section we present a subset of pub/sub protocols widely in use today and give a classification according to the defined requirements. We limit ourselves to protocols with an openly available protocol specification with open source implementations widely used. We choose the protocols AMQP, MQTT, XMPP and ZeroMQ, being protocols widely used today which meet those criteria, for further evaluation. In the following, we give a brief introduction of those protocols:

AMQP – Advanced Message Queuing Protocol (AMQP) [2] was developed as an open replacement for proprietary messaging protocols in the financial services industry. AMQP is used in popular implementations, such as RabbitMQ [34], Apache ActiveMQ [3], and Apache Apollo [4]. AMQP version 0-9-1 is an open and royalty-free specification of both a wire-level protocol and a broker model. AMQP 1.0 recently became an Organization for the Advancement of Structured Information Standards (OASIS) Standard, but includes mainly a novel wire-level protocol and only abstract broker requirements. Most functionality is defined by the broker and its behavior, which is not part of AMQP 1.0. We therefore focus on version 0-9-1, which defines the broker model most widely implemented.

MQTT – Message Queue Telemetry Transport (MQTT) [24] is a pure pub/sub protocol for constrained devices and low-bandwidth, high-latency, and unreliable networks standardized by OASIS. It has various open source implementations, such as the mosquitto broker and client library [28], the aforementioned Apache ActiveMQ [3] broker, and the Eclipse paho client library [11] There is an effort to port MQTT to sensor nodes in the slimmed down variant MQTT-S [19]. MQTT and MQTT-S are openly published with royalty-free licenses.

XMPP – Extensible Messaging and Presence Protocol (XMPP) [41] has its origin in the instant messaging protocol Jabber, motivated by the diversity of proprietary chat protocols. XMPP is based on XML streaming. The core protocol is standardized in multiple IETF RFCs and extended by XMPP Extension Protocols (XEPs), including pub/sub messaging [27]. XMPP is the protocol behind the instant messaging server ejabberd [36] and the Openfire server [21].

ZeroMQ – ZeroMQ [17] is a messaging library offering sockets with more advanced messaging patterns than Berkeley sockets. Supported patterns include request/response and pub/sub. The wire-level protocol is the ZeroMQ Message Transport Protocol (ZMTP), which is made available under the GNU General Public License. Because ZeroMQ offers an abstraction of communication principles, all required features of IoT platforms can be met by building those features on top of the messaging patterns provided. In this work, we only consider features included in the official documentation.

3.1 Taxonomy of Pub/Sub Systems

In this section, we give a classification of pub/sub protocols regarding cloud-based IoT platform requirements. An overview of our findings is given in Table 2.

In our survey, we focused on cloud-based IoT platforms with a centralized topology. All considered protocols support

such a topology, where the entire communication is done via a cloud based broker. In the face of the expected number of devices, it will be beneficial to reduce the load on the broker. In a hybrid topology the matching between producer and consumer is done by a central broker, but the actual payload is transferred directly. XMPP uses a distributed network of XMPP servers as brokers. It is also possible to establish out-of-band peer-to-peer connections using XEP-0065, corresponding to a hybrid topology. ZeroMQ can be used without a central broker in a P2P fashion. Additionally, its pub/sub sockets can be used to form a hierarchy of brokers, which forward publications and subscriptions selectively.

We observed the use of the pub/sub and request/response massaging patterns for monitoring sensing device. While AMQP and ZeroMQ support both patterns, MQTT is a pure pub/sub protocol. Request/response is still possible, but would have to use special topics on which devices or services subscribe for requests. XMPP is originally a chat protocol for direct messaging which has been extended by XEP-0060 to support pub/sub messaging [27].

We further concluded that protocols should offer advanced filtering capabilities. The considered solutions provide only topic based pub/sub with hierarchical topics. A topic-based approach is suitable for a basic subscription to a certain physical or virtual sensor. A hierarchical topic structure enables monitoring sensor sets. AMQP and MQTT support wildcards at every topic level, while ZeroMQ only supports prefix matching. XMPP uses a combination of leaf and collection nodes, where collection nodes are a set of leaf nodes, which contain published items, which is equivalent to a hierarchical topic tree. Content based pub/sub, as for triggering, is not supported.

All protocols are generally agnostic to the message format. AMQP, MQTT, and ZeroMQ use a binary encoding, permitting any format, while XMPP uses XML, which can also transport other formats, but usually at the cost of less bandwidth efficiency and increased parsing overhead. AMQP additionally offers a type system that maps transmitted data types to common data types of many languages and platforms.

It may be desirable to be given semantic guarantees in form of QoS semantics. AMQP and MQTT offer at-most-once, at-least-once and exactly-once semantics. XMPP and ZeroMQ are best-effort, but usually build upon a reliable underlying transport protocol. XEP-0079 describes additional delivery semantics including time-sensitive delivery. To supply the most recent value to late joining or temporarily disconnected subscribers, some protocols offer a last value cache (LVC). MQTT messages can be flagged as retained, the latest of which is provided to new subscribers. Nodes (topics) in XMPP may store the last value for new subscribers. Although ZeroMQ does not offer LVC directly, implementing LVC with ZeroMQ is given as an example in the official guide. AMQP does not consider LVC explicitly, however it may be provided by the broker software. There is for example a plugin for RabbitMQ that adds LVC functionality [34].

Table 2 Classification of open pub/sub middleware protocols

		AMQP	MQTT	XMPP	ZeroMQ
Pattern	Pub/Sub	✓	✓	✓[1]	✓
	Point-to-point	✓		✓	✓
Filtering	Topic-based	✓	✓	✓[1]	✓
	Content-based				
QoS Semantics	At-most-once	✓	✓	✓	✓
	At-least-once	✓	✓		
	Exactly-once	✓	✓		
	Last value caching	✓[2]	✓	✓	✓
Topology	Decentralized				✓
	Centralized	✓	✓	✓	✓
	Hybrid			✓[1]	
Binary Encoding		✓	✓		✓

4 Cloud based Performance Evaluation

While pub/sub systems offer the same core features, they differ in QoS semantics, encoding and expressiveness of filters. The effect of those differences on pub/sub performance is unknown. Furthermore, a cloud based deployment may introduce unknown effects on the performance, making a cloud based performance study necessary. This section first introduces our measurement platform and relevant metrics. It then presents the results of throughput and delay measurements conducted in the public cloud.

4.1 Measurement Setup

We expect future platforms to have one or more brokers on private or public cloud infrastructure. In our measurements, we deploy brokers on instances on Amazon's Elastic Compute Cloud (Amazon EC2) [1], which is one of the leading public cloud providers. We see gateways as clients using the pub/sub protocol to publish their data. Gateways that control actuators and user-facing applications are considered subscribers. We do not explicitly study underlying sensor network technologies, as those technologies are independent of the pub/sub protocol used. To replicate a large number of gateways and applications, we also model them using instances in the cloud, enabling large-scale experiments.

We choose to evaluate the protocols using popular implementations as representatives. Our selection of brokers given in Table 3. The broker configuration is kept as much as possible to the defaults. Erlang brokers use High Performance

[1] using XMPP Extension Protocols (XEPs)
[2] not required by standard, but mostly available via plugin

Table 3 Overview of studied publish/subscribe systems

Protocol	Broker	Version	Release date	Language
AMQP	RabbitMQ	3.4.4	2015-02-11	Erlang
MQTT	mosquitto	1.4	2015-02-18	C
XMPP	ejabberd	15.02	2015-02-17	Erlang
ZeroMQ	malamute	4.0.5	2014-10-14	C++

Erlang (HiPE). As we do not consider security, encryption is disabled.

We use a client written in the C programming language as the gateway. It publishes and subscribes using uniform interfaces to pub/sub plug-ins for every protocol. The traffic generation and logging component stays the same in every experiment. To avoid side-effects by disk IO, we buffer all results in memory before writing them to disk. We use non-persistent messages with at-most-once semantic.

Broker and publishing and subscribing processes are both deployed on EC2 instances running Ubuntu 14.04 64bit (ami-234ecc54). Brokers run on a single m3.large instance with two cores, usually of an Intel Xeon E5-2670 with 2.50GHz, and 7.5 GB of RAM. Gateway clients are deployed on m3.medium instances, with one core and 3.75 GB of RAM. We use 10 gateway instances, which host up to 100 virtual publishing gateways and up to 100 subscribing processes. The CPU load on every instance is monitored during experiments, so that bottlenecks on the client side will not be regarded as poor broker performance.

4.2 Metrics

As cloud centric IoT platforms may in the future have to support billions of devices [15], it is important that a single broker can handle as many sensor devices as possible, i.e. offers the highest throughput of messages on a given physical or virtual machine. We therefore study the *throughput*, which we define as the number of messages per second the broker can process. Additionally, the user expects fresh data from the sensor application. We thus study the *end-to-end delay* between publisher and subscriber.

4.3 Throughput Measurements

To get meaningful results for the throughput, the most crucial part is to generate realistic load from connected sensors, which will be forwarded as publications by the gateway, and realistic modeling of applications and gateways that act as subscribers.

For the measurement of throughput we decide for a well-defined, generic, reproducible, synthetic load with a fixed message size. Every gateway emits 500 messages per second of 64 bytes, which is what we consider an average size for sensor messages in traditional sensor networks. The typical maximum size of each frame in IEEE 802.15.4 is 128 bytes [20, 24]; with headers and additional layers on top, the effective payload is usually half of that, as for instance with ZigBee [24]. Every gateway has one subscriber that is interested in the whole traffic the gateway emits, so the fan out factor is 1. In the experiments, the number of publisher/subscriber pairs is increased until the pub/sub system can no longer keep up with the offered load. The maximum observed throughput is then defined as the maximum sustainable throughput.

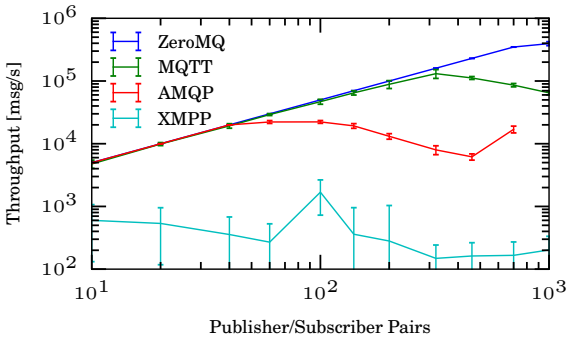


Fig. 2 Maximum sustainable throughput of the protocols.

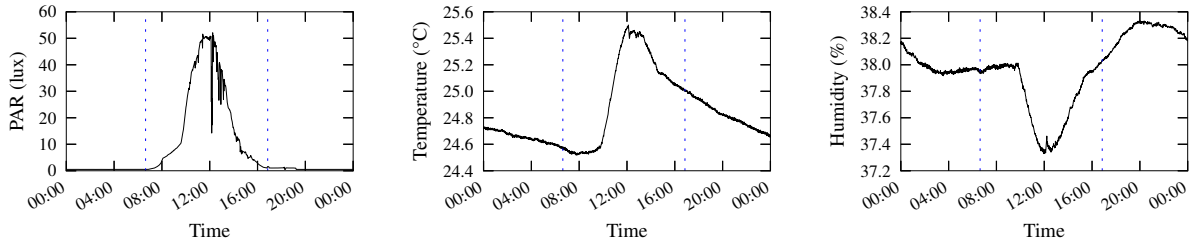
Experiments consist of generating load for 5 minutes and measuring the number of messages that reach subsribers during that time. Increased load is achieved by a higher number of publisher/subscriber pairs, which range from 10 to 1000. The increase in the number of publisher/subscriber pairs is exponential to accurately measure both very small and very high throughput values. Each parameter configuration is applied a total of 8 times.

Figure 2 shows the measured throughput when applying a certain load on the broker. The saturation of the throughput marks the individual maximum sustainable thorughput of the protocols. ZeroMQ in general has the highest throughput with almost $400\,000 msg/s$. MQTT shows a throughput of less than half of that with about $130\,000 msg/s$. AMQP has a throughput of approximately $22\,000 msg/s$. XMPP has a rather low throughput, which is mostly below $1\,000 msg/s$ and two orders of magnitude lower than the throughput of ZeroMQ.

Our ZeroMQ broker is only very basic and only prefix matching is supported. Because of that, the throughput is considerably higher than the other protocols. MQTT and AMQP offer more complex filtering, decreasing the throughput. XMPP has the largest overhead due the used XML encoding, which has to be parsed at the clients and the broker, leading to reduced throughput.

4.4 Latency Measurements

Opposed to the throughput measurements, we aim to measure the typical delay in a cloud-based setting rather than under a saturated setting. In order to investigate the latency introduced by cloud based pub/sub systems, we use a more



(a) Average measured photosynthetically active radiation

(b) Average Temperature

(c) Average Humidity

Fig. 3 Sample of average values measured on the TWIST sensor nodes over the course of one day (November 8, 2014); sunrise and sunset are marked with blue dotted lines

realistic traffic source, that replicates a real life sensor network.

We first collect representative sensor data traces in the TKN Wireless Indoor Sensor network Testbed (TWIST) [16]. Our collection is done using 90 Telos Rev. B sensor nodes [29]. Every node is equipped with a Sensirion SHT11 humidity and temperature sensor which is factory calibrated and produces digital output. Additionally, the nodes have two analogue light sensors connected to analog-to-digital converter (ADC) pins of the micro-controller: a Hamamatsu S1087 (320nm to 730nm) visible light sensor measuring the photosynthetically active radiation (PAR) and a Hamamatsu S1087-01 (320nm to 1100nm) visible to IR light sensor measuring the total solar radiation (TSR). We also collect internal voltage readings from the microprocessor.

We collect samples of the sensors mentioned above on each sensor node every second using a TinyOS application that sends the data in one packet to the serial interface that is exposed on the Universal Serial Bus (USB) port. The collection process leverages the capabilities of the TWIST testbed: each sensor node is connected to one supernode that forwards the serial packets to the TWIST server, where a trace file with all sensor values is recorded.

We collected a total of 72 hours of data, consisting of about 90 000 000 individual sensor readings. For the cloud-based delay measurements, we limit ourselves to a representative calendar day out of those traces, which is shown in Figure 3: Figure 3a shows the average of one of the light readings over all the sensors, Figure 3b shows the average temperature, and Figure 3c shows the average humidity.

The graph of the light sensors show the course of the sun on this day. After the sunset, there is a low level of remaining light until about 7:30PM from office employees still in the building. The temperature rises with solar radiation and drops after noon. The relative humidity shows a drop around noon due to the increased temperature.

We use this raw sensor data to generate sensor traces to publish. We choose to only issue a message with updated sensor data if reasonable thresholds is exceeded, i.e. if a sensor samples a value that is close to the value sampled before, the gateway will not retransmit the data, which would be reasonable in large scale sensor networks. An overview of the generated trace file is shown in Figure 4. The number of messages per second emitted follows the course of the maximum solar radiation shown before. While the mean value of messages per second is about 50, the maximum number of messages per second is almost 120.

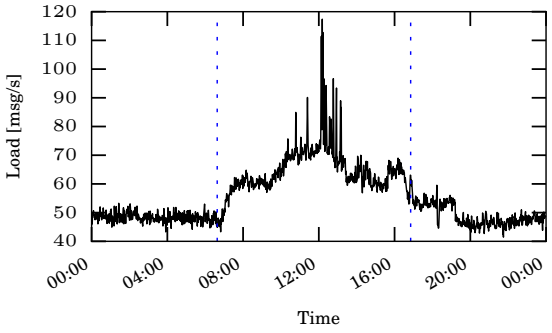


Fig. 4 Resulting load by replaying recorded trace file from TWIST testbed.

We replay collected traces in real-time, i.e. the traces are read by the test-client and published on the broker at the same time offset as they where collected. In order to get more general results, we replicate each publishing process 40 times, i.e. the equivalent of 40 sensor networks. The starting point in the trace is shifted up to 30 minutes, which is approximately the solar time difference across Germany, modeling sensor networks deployed across a mid-European country, which keeps information about the day/night cycle intact.

For modeling the subscriber side, we subscribe each subscriber process to a wildcard topic corresponding to one of the modeled gateways.

We measure the end-to-end delay, whose distribution is shown in Figure 5. The distributions of MQTT and ZeroMQ is quite similar. More than 90 % of the delay is below 1 *ms*. AMQP has in general a slightly higher delay. Still, about 80 %

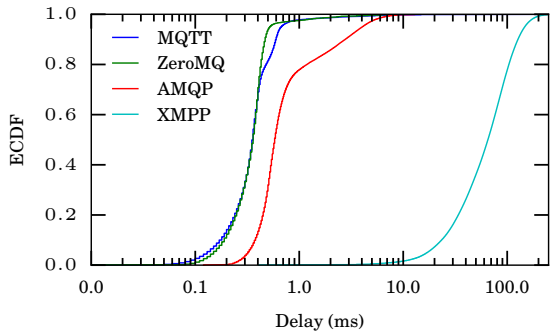


Fig. 5 Empirical distribution function (ECDF) of resulting delay by replaying recorded trace file from TWIST testbed.

of measured data is below 1 *ms*. XMPP shows a considerably higher delay up to around 250 *ms*.

We explain this behavior as follows: As we also include the network delay in our measurements, the delay is bounded by the network delay between the instances. ZeroMQ and MQTT operate very close to this minimum possible delay. AMQP is significantly more complex than the other two protocols, which would explain the higher delay. XMPP has by far the highest delay of the protocols. We expect that this is mainly due to the XML encoding of messages. While other protocols are byte-oriented, XML messages have to be parsed at least partly in order to make a routing decision. Additionally, the XML encoding makes the messages considerably larger than for the other protocols.

5 Related Work

Previous work has been done in the field of performance evaluation of pub/sub systems [39, 40, 43]. In [39], a good overview of pub/sub performance evaluation is given. In [40], the authors give a generic pub/sub benchmark based on scenarios in the field of logistics. Tran and Greenfield investigated the performance of IBM's MQSeries v5.2 with generic traffic in [43].

There are several proposals to use pub/sub systems in IoT and sensor applications [19, 38]. Sensor Andrew [38] is an IoT sensor platform using XMPP for data dissemination. MQTT-S is a stripped-down version of MQTT, which is optimized for the small frame sizes in sensor networks, which are typically under 128 bytes [19]. Gateways often found in sensor networks are used to translate MQTT-S to ordinary MQTT and relay messages to upstream brokers. Both works address the need for pub/sub integration in sensor applications but do not provide performance evaluation of their protocols.

Other studies have investigated the network behavior of Cloud-based virtual machines [37, 44, 45]. They focus on Amazon Elastic Compute Cloud (Amazon EC2) [1], which uses Xen virtualization. The CPU sharing introduces side effects such as unstable TCP/UDP throughput and abnormally large delay variations found in [44]. Findings in [37] support those results. In [45] the author show that instances of the same type have different performance characteristics that are unlikely to change with time.

Although pub/sub systems, IoT sensor applications, and Cloud-based virtual machines were studied extensively, to the best of our knowledge, there are no studies evaluating pub/sub systems in the specific context of cloud-based IoT platforms. With this study we try to answer the question which pub/sub system is best suited for those settings and what the performance impact for pub/sub operating on top of virtualized computation is.

6 Conclusion

In this work, we conducted and analysis of cloud based IoT platforms and a qualitative as well as a quantitative evaluation of four different pub/sub solutions. Although no protocol offers all features desirable in IoT settings, both evaluations showed significant differences between the protocols: Although being an extensible open protocol, XMPP cannot reach the performance observed with the other pub/sub middlewares in IoT settings. Additionally, at the server side, every stanza has to be parsed in order to make a routing decision. XMPP performs considerable worse than the other protocols regarding throughput and delay. AMQP is a full-featured MOM that offers all the building blocks necessary for creating a IoT enabled messaging broker. The performance in terms of message throughput and average delay is acceptable. However, it does not reach the performance of MQTT or ZeroMQ. MQTT is specifically designed to transport sensor-like data and has emerged to the de-facto-standard in the field. The mosquitto broker has reasonable high throughput and low delay. A crucial part also missing in MQTT is the possibility to contact connected clients directly, as MQTT is a pure pub/sub protocol. However, we believe that those shortcomings can be overcome with several minor modifications to the MQTT protocol and broker software. Our performance measurements suggest that ZeroMQ can achieve very high throughput while maintaining low delay. Also appealing is the fact that a broker is optional and a decentralized system is possible. However, ZeroMQ is no full-featured broker implementation and less expressive than the other protocols, as only prefix matching is supported. Therefore, crucial features may need to be, but on the other hand can be, added for a deployment in IoT settings.

Our conclusion therefore is twofold: We recommend ZeroMQ where a specifically tailored solution is needed and the effort of implementing missing features can be accepted. For settings where a readily available solution is necessary, we recommend MQTT, where there are a few open source software solutions that can be used out of the box.

References

1. Amazon: Elastic Compute Cloud (EC2). URL http://aws.amazon.com/ec2
2. AMQP Working Group: Advanced message queuing protocol (2010). version 0-9-1
3. Apache Software Foundation: ActiveMQ. URL http://activemq.apache.org/
4. Apache Software Foundation: Apollo. URL http://activemq.apache.org/apollo/
5. Axeda Corporation: URL http://www.axeda.com
6. Banavar, G., Chandra, T., Strom, R., Sturman, D.: A case for message oriented middleware. In: P. Jayanti (ed.) Distributed Computing, *Lecture Notes in Computer Science*, vol. 1693, pp. 1–17. Springer Berlin Heidelberg (1999)
7. Chui, M., Löffler, M., Roberts, R.: The internet of things. McKinsey Quarterly **2**, 1–9 (2010)
8. Curry, E.: Message-oriented middleware. In: Q.H. Mahmoud (ed.) Middleware for Communications, chap. 1, pp. 1–28. John Wiley & Sons (2005)
9. Davra Net.: Ruban. URL http://www.davranetworks.com
10. Digi International Inc.: Digi device cloud. URL http://www.digi.com/cloud-overview
11. Eclipse Foundation: Paho. URL https://eclipse.org/paho/
12. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys (CSUR) **35**(2), 114–131 (2003)
13. EVRYTHNG: Every thing connected. URL https://evrythng.com/
14. FIWARE: Open apis for open minds. URL https://www.fiware.org/
15. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Computer Systems **29**(7), 1645–1660 (2013)
16. Handziski, V., Köpke, A., Willig, A., Wolisz, A.: Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In: Proc. of the 2nd Int. Workshop on Multi-hop Ad Hoc Networks: From Theory to Reality (REALMAN '06), pp. 63–70. Florence, Italy (2006)
17. Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly (2013)
18. Hinze, A., Sachs, K., Buchmann, A.: Event-based applications and enabling technologies. In: Proc. of the 3rd ACM Int. Conf. on Distributed Event-Based Systems (DEBS '09), pp. 1:1–1:15. Nashville, TN, USA (2009)
19. Hunkeler, U., Truong, H.L., Stanford-Clark, A.: MQTT-S – A publish/subscribe protocol for Wireless Sensor Networks. In: 3rd Int. Conf. on Communication Systems Software and Middleware and Workshops (COMSWARE'08), pp. 791–798. Bangalore, India (2008)
20. IEEE Standard for Information technology: Local and metropolitan area networks – Specific requirements – Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs). IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003) pp. 1–320 (2006). DOI 10.1109/IEEESTD.2006.232110
21. Ignite Realtime: Openfire Server. URL http://www.igniterealtime.org/projects/openfire/
22. ioBridge, Inc.: Realtime.io. URL http://www.iobridge.com
23. Leguay, J., Lopez-Ramos, M., Jean-Marie, K., Conan, V.: An efficient service oriented architecture for heterogeneous and dynamic wireless sensor networks. In: 33rd IEEE Conf. on Local Computer Networks (LCN 2008), pp. 740–747. Montreal, Canada (2008)
24. Locke, D.: MQ Telemetry Transport (MQTT) V3.1 Protocol Specification. IBM developerWorks Technical Library (2010)
25. LogMeIn, Inc.: Xively official website. URL https://www.xively.com/. (Xively is former COSM)
26. Menzel, T., Karowski, N., Happ, D., Handziski, V., Wolisz, A.: Social sensor cloud: An architecture meeting cloud-centric iot platform requirements (2014). 9th KuVS NGSDP Expert Talk on Next Generation Service Delivery Platforms
27. Millard, P., Saint-Andre, P., Meijer, R.: XEP-0060: Publish-Subscribe (2010). URL http://www.xmpp.org/extensions/xep-0060.html. Version: 1.13
28. Mosquitto: An open source mqtt v3.1/v3.1.1 broker. URL http://mosquitto.org/
29. Moteiv Co.: Tmote sky datasheet. URL http://www.crew-project.eu/sites/default/files/tmote-sky-datasheet.pdf
30. oneM2M Technical Specification: Functional architecture. Tech. Rep. TS 118 101 V1.0.0, ETSI (2015). Version 1.6.1
31. openHAB: The open home automation bus. URL http://openhab.org
32. OpenIoT: Open source cloud solution for the internet of things. URL http://openiot.eu
33. OSIoT: Open source internet of things. URL http://osiot.org
34. Pivotal Software: RabbitMQ. URL https://www.rabbitmq.com/
35. Presser, M., Barnaghi, P., Eurich, M., Villalonga, C.: The sensei project: integrating the physical world with the digital world of the network of the future. IEEE Communications Magazine **47**(4), 1–4 (2009)
36. ProcessOne: ejabberd XMPP Server. URL https://www.process-one.net/en/ejabberd/
37. Rege, M.R., Handziski, V., Wolisz, A.: CrowdMeter: an emulation platform for performance evaluation of crowd-sensing applications. In: Proc. of the 2013 ACM conf. on Pervasive and ubiquitous computing adjunct publication, pp. 1111–1122. Zürich, Switzerland (2013)
38. Rowe, A., Berges, M.E., Bhatia, G., Goldman, E., Rajkumar, R., Garrett, J.H., Moura, J.M., Soibelman, L.: Sensor Andrew: Large-scale campus-wide sensing and actuation. IBM Journal of Research and Development **55**(1.2), 6:1–6:14 (2011)
39. Sachs, K.: Performance modeling and benchmarking of event-based systems. Ph.D. thesis, TU Darmstadt (2010). SPEC Distinguished Dissertation Award 2011
40. Sachs, K., Kounev, S., Bacon, J., Buchmann, A.: Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. Performance Evaluation **66**(8), 410–434 (2009)
41. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard) (2011). URL http://www.ietf.org/rfc/rfc6120.txt
42. Souto, E., Guimarães, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C.: A message-oriented middleware for sensor networks. In: Proc. of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC '04), pp. 127–134. Toronto, Canada (2004)
43. Tran, P., Greenfield, P., Gorton, I.: Behavior and Performance of Message-Oriented Middleware Systems. In: Proc. of the 22nd Int. Conf. on Distributed Computing Systems Workshops, pp. 645–650 (2002)
44. Wang, G., Ng, T.S.E.: The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In: Proc. of the 29th Conf. on Information Communications (INFOCOM'10), pp. 1–9 (2010)
45. Xu, Y., Musgrave, Z., Noble, B., Bailey, M.: Bobtail: Avoiding Long Tails in the Cloud. In: Proc. of the 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI '13), pp. 329–341. Lombard, IL, USA (2013)