Ann. Telecommun.
(author-created version)

Meeting IoT Platform Requirements with Open Pub/Sub Solutions

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

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 requirements of 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 pub/sub implementations: rabbitMQ (AMQP), mosquitto (MQTT), ejabberd (XMPP), and ZeroMQ. We study the maximum sustainable throughput and delay under realistic load conditions using traces from real sensors. 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.

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

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

1 Introduction

The upcoming ubiquitous network of physical objects, the Internet of Things (IoT), will offer real-time sensing of the environment and enable triggering autonomous reactions to changes in the physical world. The paradigm has led to the advent of various "smart" applications, e.g. smart city, smart enterprise and smart home, which are expected to drive the number of connected devices to billions [9, 15].

With its vast processing power, fast networking, and reliable storage, cloud computing can provide the infrastructure for provisioning, managing and controlling this large number of sensor devices as well as for big data processing of the sensed data. The flexible on-demand resource allocation and pay-as-you go cost model can enable elastic matching of the growing communication, computation and storage requirements associated with IoT applications [15].

In contrast to classical wireless sensor networks (WSN), which are tailored to and used by a single application, an added value of the IoT lies in the common usage of sensor hardware by heterogeneous applications. Constrained devices will take sensor readings only once, which will be distributed to several interested applications and services. A scalable cloud-based messaging layer can be used to tackle the challenging aspect of matching sensor data streams and interested applications or services and distribute the data accordingly.

The publish/subscribe messaging pattern enables selective distribution of messages and has become a well established data dissemination pattern, e.g. for stock market or weather data. Despite the popularity of pub/sub solutions in related applications, there is still a lack of data on the features a pub/sub system should have to meet the specific IoT requirements, the degree those features are provided by existing solutions and on the performance trade-offs of these features. In this paper, we make three main contributions towards addressing these problems:

1. We derive a set of requirements for pub/sub systems for cloud-based IoT and analyze which available open solutions satisfy those requirements;
2. We define three classes of IoT traffic based on realistic use-cases;
3. We conduct a cloud-based performance evaluation under common realistic conditions using real world trace files.

This paper is based on the talk with the same title given at the "Cloudification of the Internet of Things" 2015 conference [17]. The original work presents a performance evaluation study of the prominent messaging protocols considered suitable for IoT settings in the public cloud using four popular implementations: rabbitMQ (AMQP), mosquitto (MQTT), ejabberd (XMPP), and ZeroMQ. Metrics used are the maximum sustainable throughput under a synthetic load and the typical end-to-end delay using sensor trace files. However, the load models used only represent one specific type of sensor application, which hampers the applicability of those results for other types of IoT devices. In this paper we aim at providing more general results: While we use a similar measurement approach, we significantly expand on the considered traffic scenarios. We capture three classes of real-world IoT traffic and use them as a load to evaluate the performance of the messaging middleware under test. This work thus presents new and improved results and gives a more realistic and detailed analysis of both throughput and delay. It also gives new insights into the traffic patterns commonly anticipated in the IoT context, which was not part of the original work.

2 Pub/sub in the IoT domain

Today, several prominent academic and commercial IoT platforms share a cloud centric architecture similar to the one depicted in Figure 1 [15, 22]. Devices are connected to gateways, which forward sensor data to a cloud tier with a message broker. Additional services, e.g. data storage, analysis, or aggregation, and user facing applications connect to the broker to get access to the data. We see a trend to use a message broker using the publish/subscribe pattern to distribute the data to multiple interested applications [22]. A publish/subscribe system is a message-oriented middleware

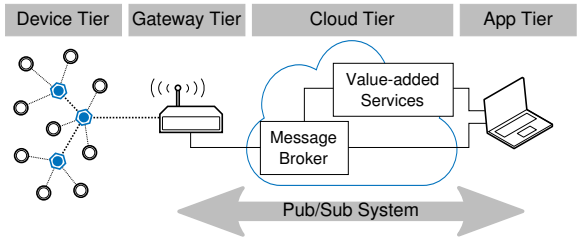


Fig. 1 General architecture of a cloud-centric IoT platform.

(MoM) [10] providing distributed, asynchronous, loosely coupled communication between message producers and message consumers. A pub/sub middleware offers three main types of decoupling [13] which makes it particularly suitable for large-scale IoT deployments: 1) Message producers (publishers) and consumers (subscribers) 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 core building block of pub/sub systems is the matching between publishers and subscribers, that may be based on different types of filtering, mostly topic or content. The filtering is usually done by multiple dedicated message brokers. 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 on the content of individual messages, e.g. if a value reaches a certain threshold predefined by the subscriber.

A similar approach dealing with large scale sensor data is stream processing. In contrast to the message-based pub/sub, stream processing applications act as complex stateful continuous queries on input streams of data generating streams of results. The focus of stream processing frameworks lies in the transformation of the input stream, whereas pub/sub focuses on the distribution of data. While we see stream processing as a promising approach to process and analyze large data streams, we argue that stream processing alone will not enable the common use of sensor hardware across multiple applications, which is one of the key properties of the IoT vision. However, we envision value-added services that use stream processing approaches on the data provided by pub/sub systems.

While pub/sub is a well established and studied messaging pattern, the use of pub/sub in cloud-based IoT settings is still not explored in detail. There is a plethora of existing solutions available without reliable data on which solution fits the specific IoT requirements best. In the following section we start addressing these problems by defining reference scenarios for cloud-based IoT deployments. We continue by giving a definition of the requirements of pub/sub systems used in cloud-based IoT deployments on which we base a quantitative evaluation of existing solutions.

2.1 Reference scenarios

Before deriving a set of requirements for generic IoT applications, we first introduce a subset of possible scenarios by defining three reference scenarios in the context of smart cities as examples:

2.1.1 Social weather service

The first use-case we envision is a social weather service, where sensor device owners connect and share their existing sensors with the public. An example is a heating, ventilating, and air conditioning (HVAC) system in an office building that takes temperature and humidity readings periodically. The system can either provide its own data publicly or use readings from other similar systems or monitor a weather forecast to improve on when to heat or cool, saving energy and improving comfort.

2.1.2 Smart car sharing

The second use-case is a car sharing system. The advent of connected cars has led to the emergence of car sharing business, renting out cars on demand. Sharing cars reduces the number of parking spots required and in turn reduce the amount of traffic looking for a parking spot. In the future, individuals could start renting out their cars in a similar manner: private cars could periodically offer themselves for rental and provide additional information, such as position, fuel level or condition. Interested parties could monitor those offers and contact the car directly to rent it.

2.1.3 Traffic monitoring

To reduce the amount of traffic, another option is to monitor traffic in the city and reroute cars on traffic jams. Cameras take pictures of traffic conditions and send them to the cloud. A cloud-based service evaluates them and assesses traffic conditions. Interested drivers or navigation units can use this information to plan their route accordingly.

2.2 Requirements for pub/sub in cloud-based IoT

This section outlines the specific IoT requirements that influence the selection of a suitable pub/sub middleware. We start with the functional requirements, that are derived from generic IoT applications and illustrated by the reference scenarios introduced above:

1. *Messaging Pattern:* All use cases require the monitoring of sensor readings. We already motivated the use of the pub/sub pattern, where symbolic addresses are used to match producer and consumer, which has to be supported. It should additionally be possible to address and contact a particular device, e.g. a car to rent. That means a point-to-point messaging pattern should also be supported.
2. *Filtering:* Interested parties usually want to receive only a subset of all information, e.g. weather sensors in the same neighborhood. The filtering capabilities of the middleware determine the expressiveness of the subscriptions a client application can issue. A topic-based approach is suitable for basic subscriptions to certain physical or virtual sensors. A hierarchical topic structure enables monitoring sensor sets. However, it often makes more sense to be informed on certain events, e.g. when a sensor reading reaches a threshold. This requires a content based pub/sub pattern or additional complex event processing. While a topic-based filtering is mandatory for cloud-based pub/sub systems, a content-based scheme is highly desirable.
3. *QoS Semantics:* While a loss of sensor data messages may be tolerable in some settings, others might require guaranteed delivery of messages, e.g. a smart car could issue alarm messages in case of theft. The middleware should therefore enable annotating subscriptions and messages with QoS requirements. Additionally, especially for sensors with low sampling rate, the system should provide subscribers with latest values while waiting for the next sensor reading.
4. *Topology:* In the context of cloud centric IoT, every pub/sub middleware must support a centralized topology, where a broker forwards the messages based on the requested filters. Please note that while we consider a single logical broker, this broker will be distributed across several physical or virtual machines. As an alternative to reduce the load on the broker, producers and consumers could communicate directly in a distributed topology, which would raise the question how to find particular sensors or actuators. We expect that most cloud-based solution would rather make use of a hybrid approach which matches producer and consumer using a central broker, while the actual payload is transferred directly.
5. *Message format:* Due to the heterogeneity of sensor hardware as illustrated in the selected use cases, it is challenging to foresee the exact format sensor data will be provided in. Pub/sub solutions must therefore be payload agnostic, i.e. make no assumptions about the payload. Further, they should support binary payloads, so that binary data serialization frameworks, such as protocol buffers [36] can be used.

These requirements will be used in the following section to derive a taxonomy of pub/sub systems. In addition to the requirements mentioned above, there are two important non-functional requirements for a cloud-based pub/sub system: 1) Due to the large number of devices expected, the system should be scalable; 2) As data may be urgent, messaging

should have low latency. We investigate these requirement separately in a performance evaluation in Section 3.

2.3 Taxonomy of pub/sub systems

Building on the general requirements for IoT platforms, in this section we present a subset of pub/sub protocols and give a classification according to the defined requirements. We limit ourselves to protocols with an openly available protocol specification with open source implementations that are in wide use. We choose the protocols AMQP, MQTT, XMPP and ZeroMQ for further evaluation. In the following, we give a brief introduction of these 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 [27], Apache ActiveMQ [4], and Apache Apollo [5]. 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 most widely implemented broker model.

MQTT: Message Queue Telemetry Transport (MQTT) [21] is a pure pub/sub protocol for constrained devices and low-bandwidth, high-latency, and unreliable networks developed by IBM and standardized by OASIS. It has various open source implementations, such as the mosquitto broker and client library [24], Apache ActiveMQ [4], and the Eclipse paho client library [12] 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) [33] 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 [23]. XMPP is the protocol behind the instant messaging server ejabberd [28] and the Openfire server [20].

ZeroMQ: ZeroMQ [18] is a messaging library offering a socket API 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. ZeroMQ is a generic messaging library enabling the addition of new functions to the core protocol. To compare the library to other systems, we only consider features included in the official documentation.

Table 1 Classification of open pub/sub middleware protocols

		AMQP	MQTT	XMPP	ZeroMQ
Messaging 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]	
Message Format	Payload agnostic	✓	✓	✓	✓
	Binary Encoding	✓	✓		✓

In the remainder of this section, we give a classification of pub/sub protocols regarding cloud-based IoT platform requirements, which is summarized in Table 1.

We motivated the use of the pub/sub and request/response massaging 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 [23].

The considered solutions provide only topic based pub/sub with hierarchical topics. 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 is not supported.

In case of critical data, semantic guarantees have to be given to the application. 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 marked to be retained. Late joining subscribers will then get the latest of those messages on the specific topics on subscribing. Nodes (topics) in XMPP

[1] using XMPP Extension Protocols (XEPs)

[2] not required by standard, but mostly available via plugin

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 [27].

We focus on cloud-based IoT platforms with a centralized topology. This means dedicated distributed brokers at a central location handle the matching and delivery of messages. All considered protocols support such a topology. In the face of the expected number of devices, it will be beneficial to reduce the load on the central brokers. 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, enabling horizontal scaling.

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.

3 Cloud-based performance evaluation

While the pub/sub systems analyzed offer the same core features, they differ in QoS semantics, encoding and expressiveness of filters, which might have subtle effects on performance. To analyze the non-functional requirements of scalability and low latency, we evaluate the systems under realistic conditions. One core aspect of those realistic conditions is the deployment in the cloud. Another aspect is a realistic replication of real-life traffic sources using a large number of devices.

3.1 Performance metrics

We choose the performance metrics according to the non-functional requirements of scalability and low latency. As cloud centric IoT platforms may in the future have to support billions of devices [15], it is important that the system can scale horizontally. This would usually be done by using a cluster of brokers. One possibility would be the sharding based on topics. For the performance of the whole system, it is however most important that a single broker can handle as many sensors and therefore sensor messages as possible, i.e.

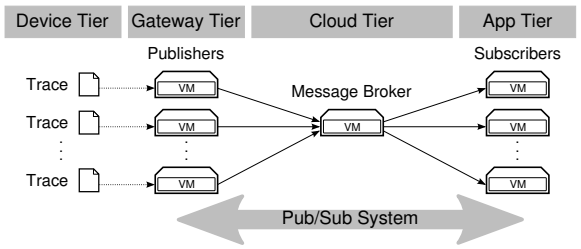


Fig. 2 Experiment setup: real-world sensor data is collected and played back on cloud instances from trace files; the broker distributes this data to subscribing processes, which are also modeled using cloud virtual machines.

Table 2 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	XPUB/XSUB	-	-	C

Table 3 Overview of Amazon EC2 instances used

	m4.large	m3.medium
OS	Ubuntu 14.04 64bit (ami-234ecc54)	
Kernel	Linux 3.13	
# virtual cores	2	1
Memory (GB)	8	3.75

it offers the highest throughput of messages on a given physical or virtual machine. We therefore study the *throughput* on a single broker, which we define as the number of messages per second the broker can process.

Another important requirement is the low latency between the measurement of sensor values and the delivery to interested applications. In this study, we do not focus on the overall latency, but only on the part of the latency that is introduced by the pub/sub system. We thus study the *end-to-end delay* between gateways and interested applications, i.e. between publishers and subscribers.

3.2 General measurement setup

In the following, we present the measurement setup we use to evaluate the selected systems using the metrics presented. Our measurements aim at evaluating cloud-based messaging broker performance as realistic as possible. An overview of our setup is given in Figure 2. As the broker is expected to run on a cloud-based virtual machine, we deploy the broker on an instance at Amazon's Elastic Compute Cloud (Amazon EC2) [1], which is one of the leading public cloud providers.

Since we cannot deploy sensor hardware at a scale stressing the broker, we follow a similar approach as Rege et al. [29] and replicate traffic sources on cloud instances. The device tier is modeled with trace files collected from real-world sensor devices. More details are given in the next sec-

tion. The trace files are used to replicate the timing behavior and message sizes of traffic seen at IoT gateways. The gateways, which act as publishers in the pub/sub system, as well as the subscribing applications, are replicated on cloud-based virtual machines.

To avoid playing the exact same trace files on different gateways and potentially biasing the results, a distribution of message sizes and inter-message times is calculated for every 15 minutes interval and played back. This keeps the information about varying message sizes and rates during the day, which would usually be correlated in a smart city, but avoids unrealistic bias introduced by artificial high correlation.

We evaluate the protocols using popular implementations as representatives. Our selection of brokers is given in Table 2. The ZeroMQ broker is using the XPUB and XSUB sockets as given in the ZeroMQ documentation and is written in C. The broker configuration is kept as much as possible to the defaults. Erlang brokers use High Performance Erlang (HiPE). As we do not consider security, encryption is disabled. Broker and publishing and subscribing processes are both deployed on EC2 instances running on the current 64bit long term support release of Ubuntu (14.04) (ami-234ecc54). The hardware is summarized in Table 3. Brokers run on a single m4.large instance with two cores, usually of an Intel Xeon E5-2676 v3 with 32 GB of RAM. Message producers and consumers are deployed on m3.medium instances, with one core and 3.75 GB of RAM. We use 10 gateway instances, which host up to 60 virtual publishing gateways and up to 60 subscribing processes each. 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.

To generate traffic and replicate subscribing processes, we use a client written in the C programming language. The application publishes and subscribes using a uniform interface that is mapped to the interfaces of the individual pub/sub systems by dedicated plug-ins. 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 with a fixed fan-out factor of 1, i.e. every publisher has one subscriber.

We use a fixed fan-out factor of 1 mainly due to two reasons: First, there is a lack of insight into the actual behavior of subscribers in an IoT context. Realistic modeling of the distribution of subscribers across topics and the timing behavior of subscribe and unsubscribe requests would be needed to accurately reproduce their behavior, which is currently not well understood. One option would be to model the behavior using stochastic processes, such as a Zipf-like distribution for the popularity of topics [8]. Another option would be to sample popular topics on real deployments, which is not easily possible without administrative access to a popular public broker. In this work, we resort to using a well-defined fan-out factor as a first step, but work on more advanced subscriber models for future work. Second, our setup simplifies the measurement of throughput and latency, since we can trivially deploy publishers and subscribers of a topic on the same cloud based virtual machine avoiding the need for precise synchronization between the processes.

We evaluate every protocol using three workload scenarios, analogous to our reference scenarios, which we define in the following.

3.3 Workload scenarios

We generalize the three reference scenarios in Section 2.1 to three main classes of sensors producing distinct traffic patterns:

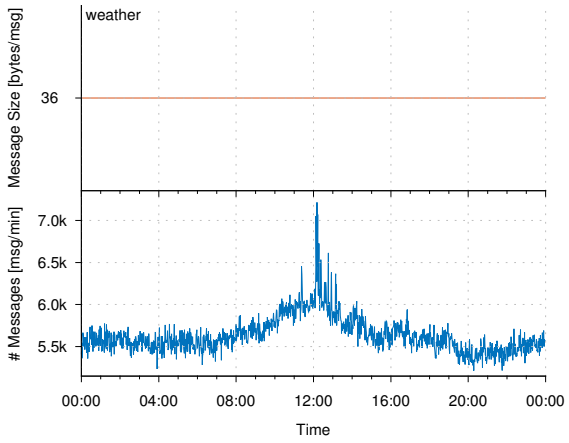
3.3.1 Simple sensors

Weather sensors are an example of what we define as a simple sensor, which samples a single value at a time, usually independent of other sensors and in a compact binary representation. Other examples could be gas, water or other resource consumption monitoring.

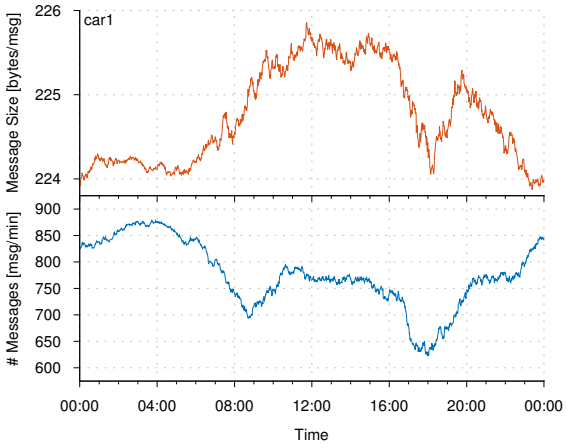
The weather data is collected upfront using the TKN Wireless Indoor Sensor network Testbed (TWIST) [16] with 90 Telos Rev. B sensor nodes [25] equipped with a Sensirion SHT11 humidity and temperature sensor, a Hamamatsu S1087 (320nm to 730nm) visible light sensor measuring the photosynthetically active radiation (PAR) and a Hamamatsu S1087-01 (320nm to 1100nm) visible to IR light sensor measuring the total solar radiation (TSR). We collect samples of the sensors mentioned above on each sensor node every second using a TinyOS[1] 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. A trace file with a total of 72 hours of data is generated.

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 are exceeded, i.e. if a sensor samples a value that is close to the value sampled before, the gateway will not retransmit the data. We consider this to be a reasonable behavior in wireless sensor networks, where energy is scarce. An overview of the generated trace file is shown in Figure 3a, which shows one day of sample data. The number of messages per second emitted varies and has its maximum around noon. Since messages are binary encoded, the message size is constant at 36 bytes.

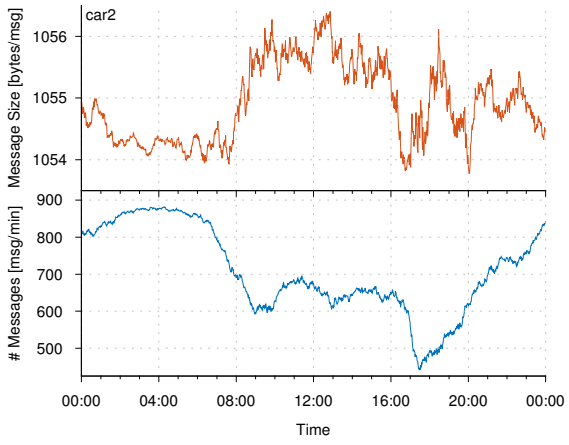
[1] `http://www.tinyos.net/`



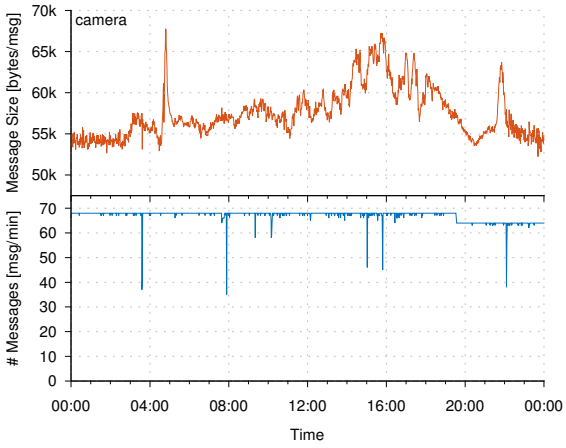
(a) Message size and rate for weather sensors



(b) Message size and rate for car sensors of business 1



(c) Message size and rate for car sensors of business 2



(d) Message size and rate for cameras

Fig. 3 Temporal characteristics of sensor tracefiles during one particular day of measurement; message size and message rate per minute are given.

3.3.2 Complex sensors

In contrast to simple readings, which are connected but also meaningful on their own, vehicle sensors are mostly only meaningful together. Considering the example of car sharing, the fuel level alone is for instance not very meaningful to potential customers without the position of the car. We define such sensors as complex sensors, which is the combination of several simple sensors, where sensor readings are grouped to form an entity of readings that are connected and send together as one message. Those messages are usually encoded with JSON or XML.

The vehicle data as a representative of complex sensor data is derived from the public websites of the two car sharing businesses [6, 11]. In the same way a browser accesses the data to draw a map of available cars, we access the publicly available data of those services for Berlin every minute and split the response up in individual status reports for every vehicle. We record the size of those individual readings and use them to replicate equivalent messages in our measurements, without storing any actual data about the cars. Trace files contain one week of data for both car sharing businesses.

Individual status reports from cars are used as messages to publish. The rationale is that every car would publish its availability along with additional information on the car, such as condition or position. One day of the data is shown in Figures 3b and 3c for each car sharing business. The change in the number of available cars follows the same general pattern and is low around 9:00 and between 17:00 and 20:00, when a lot of commuters rent cars. While car sharing business would usually get status reports either in a fixed interval or on certain events, the number of private cars available for rent from citizens would presumably follow a similar pattern. The message size is quite stable over the course of the day, but differs significantly between the two businesses.

3.3.3 Multimedia sensors

Examples of cameras as IoT sensors include surveillance camera images or video streams, but also traffic observations using cameras. Traffic camera images could be analyzed in real-time to suggest alternative routes to smart navigation systems. We define the class of multimedia sensors, which produce media, such as audio, still images or video footage. Those media streams are encoded as binary chunks of data.

As an example of media data, we use publicly available traffic cameras of the city of Berlin. In a similar manner as for the cars, we access the camera images at a fixed interval of 30 seconds and record the response size. Trace files contain one week of data for 34 cameras.

Individual images are used as messages to publish. A day of example data is shown in Figures 3d. The average message size increases towards the afternoon, probably because images with more lights will have more details that will not compress as good as dark images. The message rate is quite stable with about two images per minute per camera, as expected. Some cameras tend to not update reliably, so that we ignore all exact duplicates, explaining the slight fluctuation.

3.4 Throughput measurements

We use the traffic classes introduced before for a cloud-based measurement campaign determining the typical throughput and delay of pub/sub systems. We start measuring the throughput by applying an increasing load on the broker. This is achieved by a higher number of publisher and subscriber pairs. For the weather sensor scenario, we vary the number of publishing gateways from 10 to 600, with each gateway generating a load equivalent to 1 000 sensors, corresponding to a total of up to 600 000 emulated sensors. For the car sharing use-case, we use 10 to 600 publishers, which generate a load equivalent to up to 9.6 million individual cars. We vary the number of cameras from 40 to 600, where each publisher mimics only one camera. The increase in the number of publisher/subscriber pairs is exponential to accurately measure both very small and very high throughput values. Experiments consist of generating load for 5 minutes and measuring the number of messages that reach subscribers during that time. The load is generated according to a random position in one of the trace files. Each parameter configuration is applied a total of 8 times.

Figure 4 shows the measured throughput when the broker is loaded by the different workloads defined earlier. The saturation of the throughput marks the individual maximum sustainable throughput of the protocols in a certain scenario. Results are shown with their 95% confidence interval.

Figure 4a depicts the results for weather sensors. ZeroMQ has the highest throughput with up to $522 k msg/s$. The other protocols have a significantly lower throughput: MQTT shows a maximum throughput of less than a third of that with $134 k msg/s$. AMQP has a maximum throughput of $30 k msg/s$. XMPP's maximum of $1 k msg/s$ is about two orders of magnitude lower than the throughput of MQTT.

Figure 4b shows the results for the carsharing use-case. ZeroMQ again shows the highest maximum throughput with up to $83 k msg/s$. Notably, in this use-case, while still not reaching the throughput of ZeroMQ, the gap between the protocols has narrowed: MQTT shows a throughput of $69 k msg/s$. AMQP has a maximum throughput of $26 k msg/s$. With up to $2 k msg/s$, the throughput of XMPP is below the other protocols, but surprisingly higher than for smaller messages.

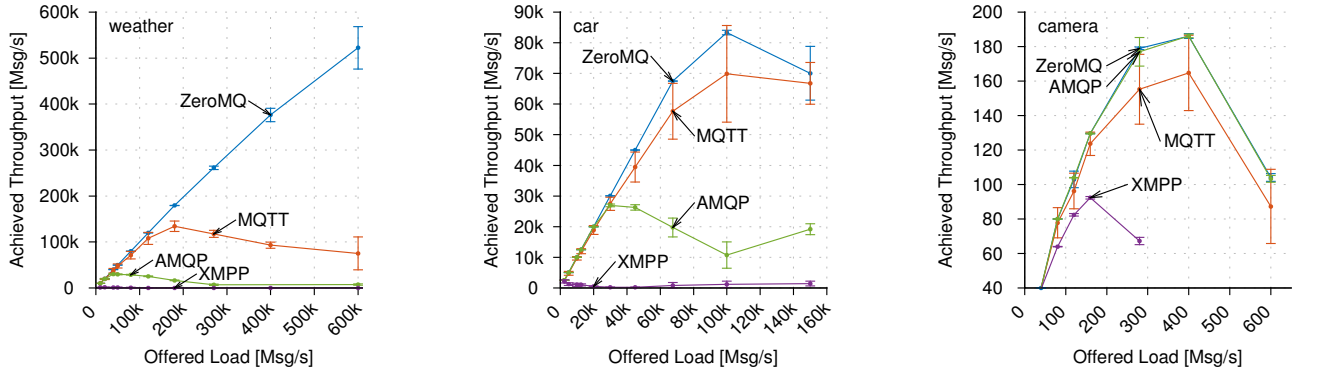
Figure 4c shows the results for camera traffic. For this use-case, AMQP has the highest throughput with a maximum of $186 msg/s$. ZeroMQ also shows a maximum throughput of up to $186 msg/s$ which is slightly below AMQP only after the decimal point. MQTT shows a maximum throughput of $164 msg/s$. XMPP shows a throughput of up to $92 msg/s$, but cannot cope at all with a load over $300 msg/s$. While for the other reference scenarios ZeroMQ was the clear leader, for the camera scenario the protocols show much more similar results.

We explain this behavior as follows: Our ZeroMQ broker is only very basic and only supports prefix matching. Because of that, the throughput is considerably higher than the other protocols especially for smaller messages. MQTT and AMQP offer more complex filtering, increasing the time spent for every filtering decision. XMPP has the largest overhead due the used XML encoding, which has to be parsed at the clients and the broker, leading to further processing overhead. For larger messages, MQTT and AMQP become more competitive, as the time spent for filtering decision is independent of the mesasge size and the time for in memory copying as well as network transmission of messages become the dominant factors of processing time.

3.5 Latency measurements

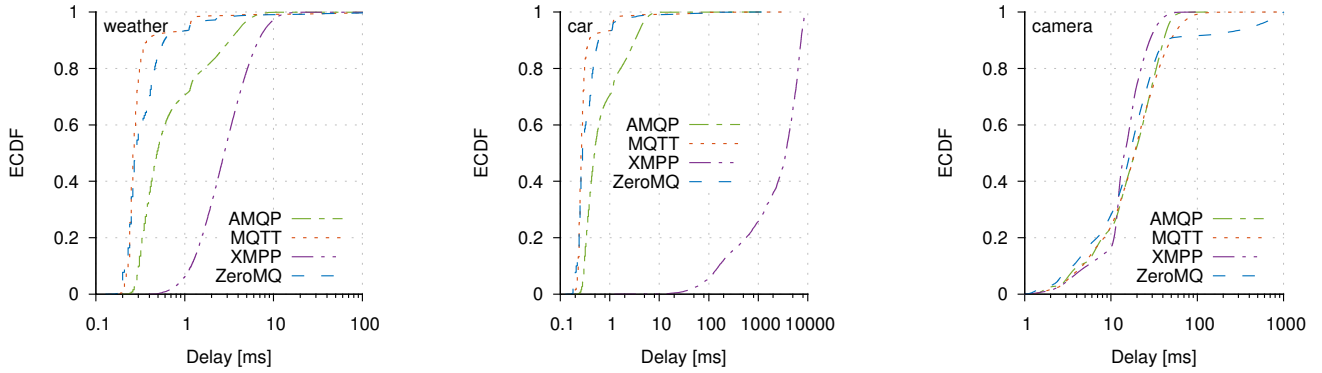
We also investigate the latency of the protocols. As the latency in a saturated setting would be unrealistically high, we observe the latency distribution over the course of one day in this separate experiment. We replicate 40 weather sensor networks with a size of 200 nodes each, which models a medium size sensor network such as the TWIST testbed. We use 100 car sharing networks similar to those recorded in Berlin. Finally, we run one camera network, similar to the one measured in Berlin with 34 cameras.

The results are presented in Figure 5. Individual plots show the empirical distribution function of the delay introduced by the pub/sub system. The graphs illustrate which percentage of successfully transmitted messages experience which end-to-end delay between publisher and subscriber.



(a) Throughput under weather sensor load. (b) Throughput under car sharing load. (c) Throughput under camera load.

Fig. 4 Throughput achieved by the protocols in the three reference scenarios.



(a) Delay under weather sensor load. (b) Delay under car sharing load. (c) Delay under camera load.

Fig. 5 Empirical cumulative distribution function (ECDF) of resulting delay by applying different loads.

Since the measured latency has a range that spans several orders of magnitude, we use a logarithmic scale to better compare results.

For the weather sensor scenario, the results in Figure 5a show that the latency distributions of MQTT and ZeroMQ are quite similar. More than 90 % of the messages have a delay below 1 *ms*. AMQP shows a slightly higher delay. Still, about 70 % of measured latency values are below 1 *ms* and 90 % below 3*ms*. XMPP introduces a considerably higher delay, but still under 3 % of the messages are delayed more than 10 *ms*.

The resulting latency distributions measured for the car sharing load are shown in Figure 5b. Although messages are significantly larger, the delay distribution of ZeroMQ, MQTT and AMQP is very similar to the previous scenario. Using MQTT and ZeroMQ, more than 90 % of measured latency values are below 1 *ms*. With AMQP, about 70 % of the messages have a delay below 1 *ms* and about 90 % below 3*ms*. XMPP shows considerably higher delays of up to 10 *s*, which indicates an overload situation.

Delay distributions for the camera use-case are given in Figure 5c. In general, the delay of all the protocols is higher than for sensors with smaller message sizes. Also, the delay distributions do not differ as significantly as for smaller message sizes. For all protocols, more than 90 % of the delay is below 100 *ms*. While achieving a lower delay more often, for ZeroMQ about 10 % of the delay is above 100 *ms*, sometimes reaching up to 1 *s*.

To summarize, the latency behavior and therefore choice of pub/sub systems is depending heavily on the distinct message sizes used in each specific use-case. While for small messages, MQTT and ZeroMQ turn out to introduce less delay, with increasing message size the delay distributions converge. The additional delay introduces by more complex filtering techniques, such as used by AMQP, becomes negligible. Also, the delay introduces by the XML encoding of the header information in XMPP is not making the overall processing time considerably higher than for other protocols in scenarios where large messages are sent.

4 Related work

The use of pub/sub systems has been proposed for a wide area of applications [13]. Classical examples of research prototypes include Hermes [26], Padres [14], and SIENA [7]. Hermes [26] is a distributed, content-based, event-based middleware combining pub/sub, peer-to-peer routing and higher-level services. In [14], the authors present Padres, a distributed content-based pub/sub middleware with features such as load balancing and network failure handling aiming at enterprise applications. SIENA [7] is an Internet-scale content-based pub-sub system, focusing mainly on the content-based routing between brokers, aiming at maximizing filter expressiveness and scalability. While this fundamental research is essential for understanding the basic concepts and techniques required for building large scale pub/sub systems, those prototypes usually do not see wide adaption in real-world IoT deployments, so are not practical candidates for a cloud-based IoT middleware at the moment.

There are several more recent proposals to use pub/sub systems specifically tailored to IoT and sensor applications [3, 19, 30, 39]. Sensor Andrew [30] 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. The more recent OpenIoT project [34] uses the CUPUS pub/sub system [3], a specifically designed content-based pub/sub with the ability to have mobile brokers, i.e. mobile pub/sub enabled gateways. They compare their system to MQTT, but only consider qualitative metrics and messaging overhead. While we focus on a cloud based architecture, Zhang, et al. [39] propose to not rely on the cloud for messaging, but instead use a fully distributed system of message routers to deliver messages across a global IoT overlay network offering a pub/sub interface. These works therefore emphasize the need for pub/sub integration in sensor applications. Still, none of them puts a focus on or provides a performance comparison of suitable protocols.

Previous work has been done in the field of performance evaluation of pub/sub systems [31, 32, 35]. However, comparisons of different pub/sub systems in the literature usually do not consider the specific requirements, use-cases or traffic patterns relevant for IoT messaging. In [31], a good overview of pub/sub performance evaluation is given. In [32], 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 [35].

Other studies have investigated the network behavior of Cloud-based virtual machines [29, 37, 38]. 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 [37]. Findings in [29] support those results. In [38] 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.

5 Conclusion

In this work, we conducted a requirement analysis for pub/sub in cloud-based IoT platforms, analyzed core features of existing open solutions and presented a quantitative evaluation of prominent representatives for each protocol under realistic load. It shows that although no protocol offers all features desirable in IoT settings, there are 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. That is presumably one reason XMPP performs considerable worse than the other protocols regarding throughput and delay. AMQP is a full-featured message oriented middleware that offers all the building blocks necessary for creating an IoT enabled messaging broker. The performance in terms of message throughput and average delay lacks behind ZeroMQ and MQTT for loads with a large number of small messages. 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 missing in MQTT is the possibility to contact connected clients directly, as MQTT is a pure pub/sub protocol. Our performance measurements suggest that ZeroMQ can achieve very high throughput while maintaining low delay, mostly independent of the load. 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 and can be added for a deployment of ZeroMQ 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 is acceptable. For settings where a readily available solution is necessary, we

recommend MQTT or AMQP, depending on the expected message sizes, as there are several open source software solutions that can be used out of the box.

Acknowledgements This work was supported by the European Regional Development Fund (ERDF) and the State of Berlin.

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. Antonic, A., Roankovic, K., Marjanovic, M., Pripuic, K., Zarko, I.P.: A mobile crowdsensing ecosystem enabled by a cloud-based publish/subscribe middleware. In: International Conference on Future Internet of Things and Cloud (FiCloud), pp. 107–114. IEEE (2014)
4. Apache Software Foundation: ActiveMQ. URL `http://activemq.apache.org/`
5. Apache Software Foundation: Apollo. URL `http://activemq.apache.org/apollo/`
6. Car2go: Berlin. `https://www.car2go.com/de/berlin/`
7. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Achieving scalability and expressiveness in an internet-scale event notification service. In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00), pp. 219–227. ACM, New York, NY, USA (2000). DOI 10.1145/343477.343622
8. Cha, M., Rodriguez, P., Moon, S., Crowcroft, J.: On next-generation telco-managed p2p tv architectures. In: Proceedings of the 7th International Conference on Peer-to-peer Systems (IPTPS'08), pp. 5–5. USENIX Association (2008)
9. Chui, M., Löffler, M., Roberts, R.: The internet of things. McKinsey Quarterly **2**, 1–9 (2010)
10. Curry, E.: Message-oriented middleware. In: Q.H. Mahmoud (ed.) Middleware for Communications, chap. 1, pp. 1–28. John Wiley & Sons (2005)
11. DriveNow: Berlin. `https://de.drive-now.com/en/#!/carsharing/berlin`
12. Eclipse Foundation: Paho. URL `https://eclipse.org/paho/`
13. 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)
14. Fidler, E., Jacobsen, H.A., Li, G., Mankovskii, S.: The PADRES Distributed Publish/Subscribe System. International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI'05) pp. 12–30 (2005)
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. Happ, D., Karowski, N., Menzel, T., Handziski, V., Wolisz, A.: Meeting IoT Platform Requirements with Open Pub/Sub Solutions. In: 1st Int. Conf. on Cloudification of the Internet of Things (CIoT'15). Paris, France (2015)
18. Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly (2013)
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. Ignite Realtime: Openfire Server. URL `http://www.igniterealtime.org/projects/openfire/`
21. Locke, D.: MQ Telemetry Transport (MQTT) V3.1 Protocol Specification. IBM developerWorks Technical Library (2010)
22. 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
23. Millard, P., Saint-Andre, P., Meijer, R.: XEP-0060: Publish-Subscribe (2010). URL `http://www.xmpp.org/extensions/xep-0060.html`. Version: 1.13
24. Mosquitto: An open source mqtt v3.1/v3.1.1 broker. URL `http://mosquitto.org/`
25. Moteiv Co.: Tmote sky datasheet. URL `http://www.crew-project.eu/sites/default/files/tmote-sky-datasheet.pdf`
26. Pietzuch, P.R., Bacon, J.M.: Hermes: a distributed event-based middleware architecture. In: 22nd International Conference on Distributed Computing Systems Workshops, pp. 611–618 (2002). DOI 10.1109/ICDCSW.2002.1030837
27. Pivotal Software: RabbitMQ. URL `https://www.rabbitmq.com/`
28. ProcessOne: ejabberd XMPP Server. URL `https://www.process-one.net/en/ejabberd/`
29. 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)
30. 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)
31. Sachs, K.: Performance modeling and benchmarking of event-based systems. Ph.D. thesis, TU Darmstadt (2010). SPEC Distinguished Dissertation Award 2011
32. 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)
33. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard) (2011). URL `http://www.ietf.org/rfc/rfc6120.txt`
34. Soldatos, J., Kefalakis, N., Hauswirth, M., Serrano, M., Calbimonte, J.P., Riahi, M., Aberer, K., Jayaraman, P.P., Zaslavsky, A., Žarko, I.P., et al.: Openiot: Open source internet-of-things in the cloud. In: Interoperability and Open-Source Solutions for the Internet of Things, pp. 13–25. Springer (2015)
35. 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)
36. Varda, K.: Protocol buffers: Google's data interchange format (2008)
37. 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)
38. 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)
39. Zhang, B., Mor, N., Kolb, J., Chan, D.S., Lutz, K., Allman, E., Wawrzynek, J., Lee, E., Kubiatowicz, J.: The cloud is not enough: Saving iot from the cloud. In: 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '15). USENIX Association, Santa Clara, CA (2015)