IoT Application Deployment Using Request-response Pattern with MQTT

Antti Luoto and Kari Systä

Tampere University of Technology, Tampere, Finland
antti.l.luoto@tut.fi, kari.systa@tut.fi

Abstract. As IoT devices become more powerful they can also become full participants of Internet architectures. For example, they can consume and provide RESTful services. However, the typical network infrastructures do not support the architecture and middleware solutions used in the cloud-based Internet. We show how systems designed with RESTful architecture can be implemented by using an IoT-specific technology called MQTT. Our example case is an application development and deployment system that can be used for remote management of IoT devices.

Keywords: Internet-of-Things, IoT, REST, MQTT

1 Introduction

We assume that devices in the Internet of Things (IoT) get more powerful and capable for complex tasks. Thus, the execution is moving towards the edge devices. This means those devices become programmable and participate in rich set of interaction with other peers in the Internet.

When IoT systems should implement functionalities of Internet systems, the implementation architectures need to be adapted to the constraints of IoT systems. In this paper we use application management as an example of a service, but the discussion is valid for many other services too. The architectures used for the management operations can either be based on technologies and approaches used in traditional Internet-based information systems or on solutions that are optimized for hardware and networking constraints of IoT. This paper explores this dilemma by showing how designs based on Internet architectures can be reimplemented on top of Message Queuing Telemetry Transport (MQTT) based IoT architecture. The work is based on earlier work [1] that demonstrates how a web-based tool can be used both for development and remote deployment of applications to IoT devices. In this system all communication between different components is based on REST [9]. Depending on the task, any component of the system may assume either client or server role. In addition, the resources in all nodes are assumed to be directly addressable with a unique address. This leads to assumption of symmetric network architecture.

The network infrastructures of IoT systems are typically asymmetric: edge devices connect to the server but the servers cannot directly contact the edge

devices. These constraints are enforced by various firewalls and Network Address Translation (NAT) systems. Thus, the REST-based system can properly work only if all devices are in the same local network.

MQTT is a lightweight protocol designed for device to device communication in IoT environments. MQTT uses a publish-subscribe (pub/sub) pattern for the communication and a centralized broker handles all subscriptions and message deliveries. Because of that, communication is limited to devices sending messages to broker and broker forwarding the messages to active subscriptions. This design is convenient if the system includes firewalls since only the broker needs to be accessible by all the components. A problem in REST-based original work [1] was that devices behind a firewall could not be accessed by other components.

Our research question is that how a REST-based system can be refactored to use MQTT in network where a firewall or NAT hides the IP address from other components. The core technical challenge is how to convert the request-response pattern assumed by REST to the pub/sub protocol of MQTT.

The rest of the paper is structured as follows. In Section 2 we introduce MQTT more in detail, present a comparison of MQTT and HTTP and suggest a solution to implement request-response in MQTT. In Section 3 our proof of concept is described and compared to the original HTTP work. In Section 4 we evaluate the work. After describing our work, we compare our work to the work made by others in Section 5. Finally, in Section 6 we provide some concluding remarks and thoughts for the future work.

2 Mapping of HTTP Concepts to MQTT

MQTT uses pub/sub communication pattern [20]. This means that senders do not send messages directly to recipients. Instead, the messages are just published for possible receivers. Similarly, the receivers express interest by subscribing to forthcoming messages. MQTT includes a special *broker* component which manages the subscriptions and publishing of the messages. To direct messages to intended recipients, the subscriptions and publishing in MQTT are directed to *topics* which may form hierarchical structures. Topics are constructed so that the different levels in the hierarchy are separated by a slash character. For example, if `abc` is a first-level topic then `abc/123` is a second-level topic. The subscriber can use wild cards to subscribe to multiple topics in the hierarchy. A special character `+` is used as a single-level wild card and `#` character is used as a wild card for multiple levels. For example, a subscriber of topic `abc/+` gets messages sent to topics `abc/123`, `abc/xy`, and to all other topics that start with `abc/` and also have only one additional level. Since our aim is to refactor a REST-based architecture to work with MQTT, we present a brief comparison between the main features of both approaches. The comparison is constructed similarly to the one made about REST and actor model in [17]. The comparison is summarized in Table 1.

Resource/MQTT client. Both HTTP resources and MQTT clients subscribing to messages can be seen as individual and isolated entities. In addition,

they are both the fundamental parts of each approach which also define the used vocabulary for the service. For the most part, the information available in the system and the domain functionality is encapsulated in these concepts.

URL/MQTT topic. In REST all resources have a unique resource identifier (URL) that is accessed with the CRUD operations. In HTTP systems this means that an address is a combination of a host address and the path to the resource within the host. In MQTT the end-point addressing is replaced with topic hierarchies. MQTT allows, but does not enforce, designs where each resource has a dedicated topic (tough there is a danger that someone else has subscribed to that topic). Thus, the topic hierarchy in MQTT can be constructed by copying the URL structure of the corresponding REST-based architecture. Even the syntax – slash (/) used as a separator – is similar. So, as an example `http://example.com/abc/123` can be presented as a three-level-topic `example.com/abc/123` in MQTT.

HTTP request and response/MQTT message. HTTP communication is based on request-response paradigm where each request gets a response that contains at least a status (for example 200, 202, 404) and optionally also content as a payload of the response. In addition, the messages follow the uniform interface and apply standard operations on the resources – for example POST, GET, PUT and DELETE. In MQTT the messages are unidirectional and MQTT clients do not get any responses. Thus, if the application depends on a response, a separate response message has to be sent to the orginal sender. Furthermore, the messages in MQTT do not have standard types like HTTP.

Statelessness/With or without state. HTTP is a stateless protocol. There are some mechanisms in MQTT that retain information about the state of the client (for example *persistence* that keeps some information about the client on the broker in case of a lost connection) but using those is not mandatory. MQTT can be used as a stateless protocol when needed.

Client-Server/Publisher-subscriber. In HTTP the requests are always sent to a known server and resource. Although any entity can take either client or server role, each message is sent by a client to a dedicated server. MQTT has also an asymmetric communication pattern, but the client cannot address the message to a certain subscriber.

Table 1. The counterparts of HTTP (left) and MQTT (right).

Resource	MQTT client
URL	MQTT topic
HTTP request and response	MQTT message
Statelessness	With or without state
Client-Server	Publisher-subscriber

In using pub/sub architecture instead of HTTP-based REST, two fundamental issues need to be solved: design of an addressing mechanism that matches the URL-structure of the REST architecture, and implementation of the response

mechanism that the application layer assumes. To answer these issues, the topics and contents of the messages have to be designed so that the response messages are directed to the original sender and matched to the original message. Fundamentally there are two places to encode the required information: message content and the topic hierarchy. Many combinations of those can be used to implement responses. We give two examples.

(1) Before sending any request, the caller subscribes to a unique topic for the response for the forthcoming request. In this case, the content of the response message consists of a status code and a payload assumed by the application. The topic hierarchies need to designed so that the topic for the reply message can be derived from the request automatically. Unsubscription from the response topic needs to be done so that the number registered topics in a long-living system stays limited.

(2) Each caller has a generic response topic that it subscribes to, and all responses to that caller are sent to that topic. The response message needs to include identification information about the original message and the client needs to match the response to the correct request. This option moves a part of the responsibility of directing a response to the correct request from the broker to the client. The broker is not assumed to interpret application-specific content of the response topic. If an application creates multiple requests - and they may be active simultaneously, then the matching of the response to correct topic becomes even more complicated. Compared to the previous option, a smaller number of topics is needed and they are not created and removed dynamically.

After analyzing the options, we selected the first option since it requires the least modifications to our application code, and since the amount of devices in IoT systems is expected to grow, a flexible topic structure is beneficial. The downside is the need for creating response topics dynamically and subsequently a need to remove them dynamically, too.

3 Proof of Concept

3.1 Original System

The original system consists of three active components: Integrated Development Environment (IDE), runtime environment and Resource Registry (RR). The IDE runs on web browser, and it is used for programming, deploying and managing the applications on devices. The runtime environment is pre-installed on the participating IoT devices and it essentially makes devices small application servers. The runtime environment, that is implemented with node.js [22], provides a REST API for installing, starting, stopping and removing applications. The runtime environment can execute multiple installed applications simultaneously. All devices and installed applications need to register themselves with the RR that maintains information about them. RR also provides an API for discovery of devices, device capabilities, installed applications and services provided by the applications. The architecture of the system is shown on the

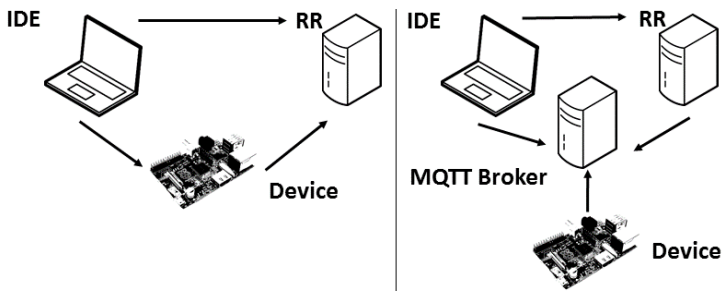

Fig. 1. Left: the original framework with HTTP. Right: HTTP replaced with MQTT.

left in Fig. 1. The arrows depict communication between the components: IDE deploys and manages applications in devices, devices register themselves and applications to RR, and IDE queries devices and applications from RR. All the communication in the original system is implemented using HTTP. Further information about the original system can be found from [1] and [14].

3.2 Motivation: Network Limitations

The original system works well if all components are in the same network and there are no firewalls or NAT systems between the components. In reality the networks impose limitations. For example, the devices are often behind NAT and firewall for security or other reasons. We have experienced these problems in practice when demonstrating the system outside of our laboratory network. Our RR and many devices run in our research network, but we have wanted to bring the IDE and some example devices to remote locations where the connection is based on local wireless networks or portable 4G access points. Due to the limitations of those networks, some communication presented on the left in Fig. 1 was not possible because the devices were not addressable from the server-side components located in the university premises.

We wanted to see if MQTT can solve these problems. In addition, we have learned from IoT practitioners that MQTT is a rising technology in IoT domain and thus we wanted to test if our systems can be made MQTT compatible. Thus, we wanted to port our system to MQTT-based communication to ensure that our earlier research results can be applied under realistic network configurations, and to make demonstration of our system easier.

3.3 MQTT Implementation

We use MQTT.js [19] library and Mosquitto [18] MQTT broker. We implemented the communication from IDE to devices and from devices to RR seen in Fig. 1 with MQTT. Only communication that requires interaction between IDE to RR has been left using HTTP. We assume that RR as the central server is accessible from all components of our system. The resulting architecture is presented on

the right in Fig. 1. We think that communication from IDE to devices and from devices to RR is enough for a proof of concept to show MQTT working in our use case. However, we do not foresee any problems in implementing communication from IDE to RR with MQTT as well.

From the different options to implement the request-response pattern with MQTT presented in Section 2, we selected the one where a unique topic is created for every request and response. The reason is that we want the broker to handle the directing of the messages so that the clients do not have to receive extra messages and decide what to do with them. It also requires less changes to our system. The topic hierarchies designed for our system are presented in Fig. 2. One notable detail in the hierarchies is the relation of replies to corresponding requests. In our solution each request is given a unique identification (rID) and a reply-topic for each rID is created before the request. Device identifications (dID) are used as unique identifiers for devices.

The left hierarchy in Fig. 2 consists of the following structure. The first level 'device' is a topic describing the problem domain - we are deploying applications to devices. Each device in the system has a separate branch which is identified with a unique dID. For example, a device with an identification XX has a topic starting with `device/XX`. The devices manage their own branches that direct the messages to them. Essentially this implements the addressing scheme discussed in Section 2: the beginning of the topic (`device/<dID>`) corresponds to IP address (domain name) and the rest corresponds to the path of URL. For details, see Table 2 where a mapping between some URLs of the system and MQTT topics is given. The level 'app' of each dID branch is for applications installed on the devices. The branches following from here are for requests and replies addressed to the applications. It would be easy to extend the hierarchy by adding a level for application IDs. For example then it would be possible to address a certain application with a topic (`device/<id>/app/<appID>`).

The topics enabling communication with RR are seen on the right in Fig. 2. The first level indicates that the hierarchy is meant for RR. The topics `RR/request` and `RR/reply` are used for registering devices. Device identification is not used yet because unregistered device does not have a dID yet. On the second level each registered device has a separate branch identified by dID. 'Apps'

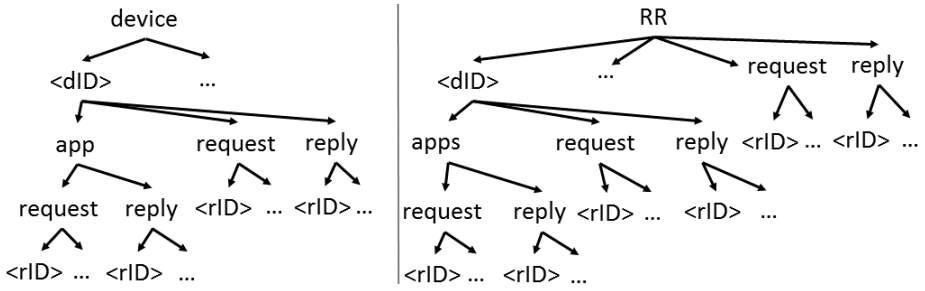

Fig. 2. Topic hierarchies for MQTT implementation. Left: device. Right: RR.

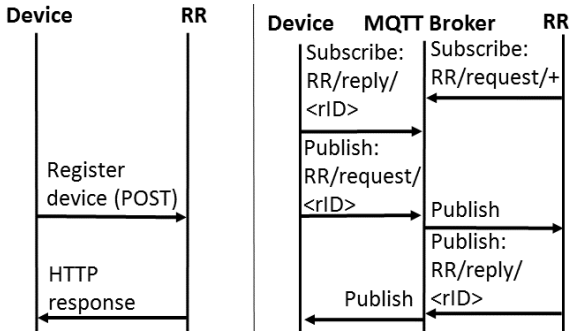

Fig. 3. Left: register a device with HTTP. Right: register a device with MQTT.

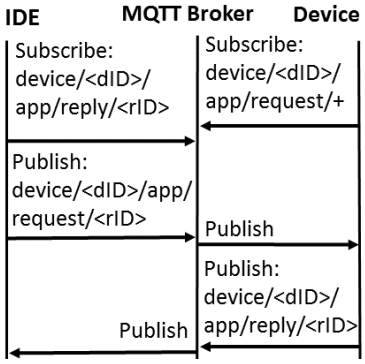

Fig. 4. An application is deployed to a device.

branch is used for managing the applications running on the device or retrieving information about them. For example, it is used by a device to publish the current state of all its applications to RR using (`RR/<dID>/apps/request/<rID>`).

The sequence charts in Fig. 3 describe how a HTTP request-reply relates to MQTT request-reply. One can think that logically HTTP POST and HTTP response correspond to one pair of MQTT publications but in order to do the publishing, response subscriptions for both the parties need to be made. The sequence in the Fig. is about device registering itself to RR. For the RR to receive registrations, it needs to subscribe to a topic for RR related requests with a wild card (+). The device in turn, subscribes to a topic where it knows to expect the response. Then the device sends a registration request to a topic that contains a unique identifier for the request and is also subscribed by RR with the wild card. RR will then send a response to the response topic subscribed by the device. The status code in the reply tells whether the registration was successful or not. After the registration, the device can unsubscribe from the response topic.

Table 2. Mapping of the URLs to MQTT topics when a device sends requests to RR.

URL	**MQTT topic**
RR-host-address	RR/request/<rID>
RR-host-address/<dID>/apps/<appID>	RR/<dID>/apps/request/<rID>

In Fig. 4, the situation is similar to the device registration case with the difference that the communication happens from the IDE to a device. In this case, a user deploys an application using the IDE. Initially the device (that has already been registered and thus has a dID) must have subscribed to a topic that is used for deploying applications to that device. Before sending the deployment message, the IDE subscribes to a topic dedicated for the response sent by the device. During the deployment, the IDE first creates a unique rID, subscribes to a reply topic of that rID and publishes a message to the topic that the receiving device has subscribed to. After completion of the deployment, the device publishes the response to the unique topic expected by the requesting IDE. The response contains information whether the deployment was successful or not. The IDE can unsubscribe from the reply topic finally.

4 Evaluation

The proof of concept works as expected and allows demonstrations outside of our lab. The original demo did not work if IoT devices were behind a firewall because IDE could not access the devices directly. The MQTT implementation has a centralized broker that is accessible from everywhere. As a result our demonstrations worked as expected. The required changes to the original system were limited and local. The adaptation to MQTT was done by replacing the source code that sends the HTTP requests with a source code that publishes an MQTT message and subscribes to the reply.

The next code example show how the original Javascript code snippet using HTTP is refactored for MQTT. The operation shown in the example is a device registering to RR The original code simply sends an HTTP POST to a URL hosted by RR and saves the returned dID for later use. The handling of the response in refactored code is a bit more complex because it needs to be converted to string and the response status and message body are not automatically parsed. While the snippet is an example from our case, the solution can be generalized so that for using MQTT in request-response style, dedicated request-reply topics with request IDs offer one flexible solution. The solution also provides the other benefits of MQTT such as lower resource consumption and ability to use normal pub/sub pattern when needed.

```
var options = {uri: RRInfo.url, method: 'POST', json: deviceInfo};
request(options, function(err, res, body) {
  if(!err && res.statusCode == 200) {
```

```
    //Read dID from HTTP response
    resourceRegistryInfo.idFromRR = body.toString();
    //Save dID to config file... (removed from this snippet)
  } else {//Error handling...}
});
//****The previous code refactored to use MQTT****
//Subscribe for response and register the device. Use unique request ID.
client.on('connect', function () {
  client.subscribe('RR/reply/' + rId);
  client.publish('RR/request/' + rId, JSON.stringify(deviceInfo));
});
//Listen for responses and parse the status code from the body.
client.on('message', function (topic, message) {
  if(topic == 'device/reply/' + rId) {
    if (message.toString().substr(0, 3) == '200') {
      //Read dID from MQTT response. It is after the status code.
      deviceInfo.idFromRR = message.toString().substr(5, message.length-1);
      //Save dID to config file... (removed from this snippet)
      client.unsubscribe('RR/reply/' + rId);
    } else {//Error handling...}
  }
});
```

We did not discover any performance issues while testing. The maximum size of an MQTT message is about 256 Megabytes and the size of the messages in our system has been under 10 kilobytes. Temporary topics are removed by the requester after receiving the response which increases the amount of traffic a bit but prevents the system from growing memory usage continuously. The method uses complex topics though simple and short topic structures could use less resources [13]. On the other hand, a flexible topic structure is important when adding new features. Still, since MQTT is IoT optimized, using it should help saving the resources when compared to HTTP. Our preliminary results with 1000 request-replies suggests that MQTT uses less CPU time and memory even with request-reply pattern but the detailed analysis is left for future work.

Robustness and reliability requires future work. In HTTP-based systems the clients either get a response or an error condition. In the current MQTT system conditions like network errors may lead to situations where client never gets any notice and the requesting subsystem has no way to discover what went wrong. The quality of service (QoS) of MQTT could provide partial solution but most probably some additional logic needs to be added. The QoS level used by us is MQTT level "zero" which means that there are no guarantees of delivery of the messages. However, we have not discovered any cases of undelivered messages in our test network.

If extra security is needed in REST, there are standard mechanisms for encryption, authentication and authorization. In the pub/sub systems basically anybody can subscribe and publish. This increases risks if all entities in the system are not trusted. MQTT also provides security features such as authentication and authorization but those are not used in this work.

5 Related Work

There is some research on making request-reply over pub/sub architecture [6][12][25] but they are on general level rather than solutions for specific network architecture problems in IoT. More recent studies and techniques take IoT into account. A draft document by Advancing Open Standards for Information Society [23] addresses the problem of using MQTT for request-response pattern by stating that request-response is needed when IoT device reads data from server/other device or vice versa, and when IoT device needs to set a value in server/other device or vice versa with a confirmation that the operation was successful.

Open Mobile Alliance Lightweight Machine-to-Machine (OMA LWM2M) is an IoT protocol that supports device and application management [24]. It has overhead for our purposes since it has an object and resource model that we do not need (we already have a resource model) and it uses CoAP that can not communicate to parties behind NAT as easily as MQTT. While tunneling, port forwarding or particular connection requests can be used with CoAP [15], we thought MQTT is more simple and enough for our needs. Several authors [10][3][27] mention the benefits of MQTT when communicating beyond NAT.

There is also some research about the relation of MQTT and REST. Collina et al. [5] studied a broker that bridges MQTT and REST by exposing MQTT topics as REST resources and vice versa, so that it is possible to use MQTT via REST but they do not try to use MQTT similarly to REST. Chan and Liu [4] implemented an MQTT proxy in their REST architecture comparing latency and performance between the protocols. They do not discuss how to implement functionality similar to REST with MQTT.

Some tool tutorials [11] (visual Java programming environment), [2] (example with Emitter.io broker), [7] (Java based IoT framework) [26] (complex messaging middleware) describe solutions which use request-reply over MQTT. However, we did not see enough benefits to utilize such tools with our approach of minimal refactoring. Other techniques to implement the functionality exist as well. HTTP long polling could work but we wanted to use MQTT because it should support IoT better. HTTP is not designed for pushing data and thus it is not as efficient [21]. Websockets could be another alternative often used in web browsers to create full-duplex communication. However, websockets are not designed for constrained devices and do not support IoT domain well [16].

Pulga [8] is an MQTT broker, targeted to be run on low-resource devices, that can deploy binary applications. In contrast, we use broker that is installed on a desktop server and deploy source code. However, there seems to be relatively few scientific publications about using MQTT in request-response (or REST) style.

6 Conclusions and Future Work

We presented a work where originally REST-based communication was substituted with MQTT by using it in request-response manner. The use case of the work was an IoT development and deployment framework presented in [1]. The

initial motivation for the work came from the problems of using HTTP to access devices that are behind a firewall. Another reason is that MQTT is more suitable for IoT devices with limited resources, and MQTT is used in practical IoT implementations. The main technical challenges in our research were related to the implementation of request-response paradigm. Our solution is based on separate response message and design of topic hierarchy with specific request and reply topics. In addition, a status code needs to be added to the content of the response. Nevertheless, the similarities between the concepts of REST and MQTT helped us in the design work.

One way to extend the work would be to implement the whole system to support MQTT. Currently, only the communication that prevented us from demonstrating the system remotely have been implemented with MQTT. For example, the IDE still uses REST for communicating with the RR. The security aspects require further analysis and research since the current implementation assumes that all participating entities trust each other. By adding authentication and encryption technologies the security could be improved. In addition, the system should be evaluated with a larger scale set-up and amount of data. The number of messages, required processing and energy consumption should be measured to answer the questions raising from the growing IoT phenomenon.

References

1. Ahmadighohandizi, F., Systä, K.: Application development and deployment for iot devices. In: CLIoT 2016 : The 4th Workshop on CLoud for IoT (2016)
2. Atachiants, R.: Stock Explorer: Using Pub/Sub for Request/Response. https://www.codeproject.com/Articles/1159256/Stock-Explorer-Using-Pub-Sub-for-Request-Response (2016), accessed: 2017-02-03
3. Bellavista, P., Zanni, A.: Towards better scalability for iot-cloud interactions via combined exploitation of mqtt and coap. In: Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), 2016 IEEE 2nd International Forum on. pp. 1–6. IEEE (2016)
4. Chen, H.W., Lin, F.J.: Converging MQTT resources in ETSI standards based M2M platform. In: Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE. pp. 292–295. IEEE (2014)
5. Collina, M., Corazza, G.E., Vanelli-Coralli, A.: Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In: 2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications-(PIMRC). pp. 36–41. IEEE (2012)
6. Cugola, G., Migliavacca, M., Monguzzi, A.: On adding replies to publish-subscribe. In: Proceedings of the 2007 inaugural international conference on Distributed event-based systems. pp. 128–138. ACM (2007)
7. Documentation for Eclipse Kura: MQTT Namespace Guidelines. https://eclipse.github.io/kura/ref/mqtt-namespace.html#mqtt-request/response-conversations (No date), accessed: 2016-10-05
8. Espinosa-Aranda, J.L., Vallez, N., Sanchez-Bueno, C., Aguado-Araujo, D., Bueno, G., Deniz, O.: Pulga, a tiny open-source mqtt broker for flexible and secure iot

deployments. In: Communications and Network Security (CNS), 2015 IEEE Conference on. pp. 690–694. IEEE (2015)
9. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
10. Fremantle, P.: A reference architecture for the internet of things. WSO2 White Paper (2014)
11. Gunawan, L.A.: Request/Response Pattern Over MQTT. http://www.bitreactive.com/mqtt-request-response/ (2014), accessed: 2016-10-05
12. Hill, J.C., Knight, J.C., Crickenberger, A.M., Honhart, R.: Publish and subscribe with reply. Tech. rep., DTIC Document (2002)
13. HIVEMQ: MQTT Essentials Part 5: MQTT Topics & Best Practices. http://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices (No date), accessed: 2016-10-05
14. Hylli, O., Ruokonen, A., Mäkitalo, N., Systä, K.: Orchestrating the internet of things dynamically. In: To appear in first International Workshop on Mashups of Things and APIs (MoTA) co-located with MIDDLEWARE 2016 (2016)
15. Jaffey, T.: MQTT and CoAP, IoT Protocols. https://eclipse.org/community/eclipse_newsletter/2014/february/article2.php (2014), accessed: 2017-02-03
16. Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F., Alonso-Zarate, J.: A survey on application layer protocols for the internet of things. Transaction on IoT and Cloud Computing 3(1), 11–17 (2015)
17. Kuuskeri, J., Turto, T.: On Actors and the REST. In: International Conference on Web Engineering. pp. 144–157. Springer (2010)
18. Mosquitto: Mosquitto - An Open Source MQTT v3.1/v3.1.1 Broker. https://mosquitto.org/ (No date), accessed: 2016-10-05
19. MQTT.js: The MQTT client for Node.js and the browser. https://www.npmjs.com/package/mqtt (No date), accessed: 2016-10-05
20. MQTT.org: MQTT. http://mqtt.org/ (No date), accessed: 2016-10-05
21. Nicholas, S.: Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android. http://stephendnicholas.com/archives/1217 (2012), accessed: 2016-10-05
22. Node.js: About node.js. https://nodejs.org/en/about (No date), accessed: 2016-11-08
23. OASIS MQTT Technical Committee: Request/Reply Message Exchange Patterns and MQTT Version 1.0 Working Draft 02. https://www.oasis-open.org/committees/download.php/56280/reqreply-v1%200-wd02.docx (2015), accessed: 2016-10-05
24. Rao, S., Chendanda, D., Deshpande, C., Lakkundi, V.: Implementing lwm2m in constrained iot devices. In: Wireless Sensors (ICWiSe), 2015 IEEE Conference on. pp. 52–57. IEEE (2015)
25. Rodríguez-Domínguez, C., Benghazi, K., Noguera, M., Garrido, J.L., Rodríguez, M.L., Ruiz-López, T.: A communication model to integrate the request-response and the publish-subscribe paradigms into ubiquitous systems. Sensors 12(6), 7648–7668 (2012)
26. Solace Systems: Request/Reply (MQTT). http://dev.solacesystems.com/get-started/mqtt-tutorials/request-reply_mqtt/ (No date), accessed: 2016-10-05
27. Uehara, M.: A case study on developing cloud of things devices. In: Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on. pp. 44–49. IEEE (2015)