IET Software

Special Issue: Software Architecture for the Web of Things (SAWoT)

IET Journals
The Institution of Engineering and Technology

Fighting network restrictions of request-response pattern with MQTT

ISSN 1751-8806
Received on 29th September 2017
Revised 10th February 2018
Accepted on 29th March 2018
E-First on 10th May 2018
doi: 10.1049/iet-sen.2017.0251
www.ietdl.org

Antti Luoto[1] ✉, *Kari Systä*[1]

[1]*Laboratory of Pervasive Computing, Tampere University of Technology, Korkeakoulunkatu 10, FI-33101 Tampere, Finland*
✉ *E-mail: antti.l.luoto@tut.fi*

Abstract: As Internet-of-Things (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. The authors show how systems designed with RESTful architecture can be implemented by using an IoT-specific technology called message queuing telemetry transport (MQTT). Their example case is an application development and deployment system that can be used for remote management of IoT devices. To evaluate the proposed solution, they performed resource consumption experiments to compare HTTP and request-response usage of MQTT. The results suggest that MQTT uses less central processing unit time and memory.

1 Introduction

We assume that when the devices in the Internet of Things (IoT) become more powerful and capable of performing complex tasks, the execution will move towards the edge devices. This means the devices become programmable and participate in rich interactions with other peers on the Internet.

However, the network and system architectures of IoT systems are usually constrained and optimised for minimal consumption of energy and other resources. One typical constraint is that devices are connected to the local network and that network is connected to the Internet through a gateway. This kind of network configuration imposes constraints, for example, the devices are not directly accessible from services on the Internet.

In this article, we explore compatibility between IoT and Internet architectures. The research was inspired by our existing system [1] that uses the RESTful architecture for device management and the management system has to address devices directly. In this research, we show how the functionality of the original system can be preserved even if the network configuration imposes constraints.

Concretely, this study explores the issues between Internet architectures and constraints in IoT systems by showing how designs based on RESTful Internet architectures can be refactored on top of message queuing telemetry transport (MQTT). The work is based on earlier work where a web-based tool can be used for both development and remote deployment of applications to IoT devices. While evaluating and demonstrating this system, we realised that the devices behind a network gateway could not be accessed by other components. The original architecture of the system is based on RESTful style and uses HTTP in all communications. Depending on the task, any component – including devices – of the system may either offer or consume resources. This architecture could not work under the network constraints.

MQTT is a lightweight protocol designed for device-to-device communication in IoT environments. MQTT uses publish–subscribe pattern for the communication and a centralised broker handles all subscriptions and message deliveries. Due to that communication is limited to devices sending messages to the broker and the broker forwards the messages to active subscriptions. This design is convenient in the constraining network configurations since only the broker needs to be accessible by all the components. In addition, MQTT was assumed to use fewer resources than HTTP used by REST.

Our research question is that how a REST-based system can be refactored to use MQTT in network configurations where the real internet protocol (IP) addresses of all components are not visible to the other peers in the system. The sub-questions include how big changes are needed in the implementation and what are the performance characteristics of the two options. The core technical challenge is how to convert the request-response pattern assumed by REST to the publish-subscribe pattern of MQTT version 3.1.1.

The proposed solution uses separate response messages and a flexible MQTT topic hierarchy with specific request and response topics. The solution is designed for an IoT system consisting of multiple devices and it fits our case with a relatively small amount of code refactoring. Additionally, we compared the resource consumption of request-response styled MQTT usage with HTTP. The comparison suggests that MQTT uses less central processing unit (CPU) time and memory.

The rest of the paper is structured as follows. In Section 2, we describe the problem domain and introduce the basics of MQTT. In Section 3, we compare our work with the work performed by others. In Section 4, we discuss a solution to the mismatch of MQTT and REST and present our proof of concept. In Section 5, we evaluate the work with a source code example and describe the resource usage experiments. In Section 6, we discuss the usefulness of the solution. Finally, in Section 7, we provide some concluding remarks and thoughts for future work.

2 Background

2.1 Realistic IoT architectures

According to Datta *et al.* [2], a recent literature about IoT architectures fails to address real life problems related to 'device and service discovery, controlling endpoints from mobile clients and interaction with endpoints through standard protocol'. To overcome this challenge, Datta *et al.* proposed a concept of 'realistic IoT architecture'. It consists of machine-to-machine (M2M) devices and endpoints, a wireless gateway with web services and mobile clients. In such architectures, the network elements, like firewalls and network address translations (NATs), impose restrictions on applications. Almost every organisation and private home user connected to the Internet has some kind of firewall and NAT today.

Firewall is a security mechanism for filtering, validating and blocking the network traffic. It protects the devices by separating them from the other parts of the Internet. NAT is a method, implemented in firewall or gateway, which converts IP address to another to save IP address space and to increase security by hiding the local network from the public one. On the other hand, NAT also causes restrictions in cases where the services in the Internet or backend need to connect devices in the local network.

We experienced such problems with our case system that used RESTful HTTP-requests from server-side to devices. This happened, for example, when we demonstrated the system outside of our laboratory network. Some of the devices were located in our research network, but we needed to bring the development interface and some example devices to remote locations. There the connection was usually based on local wireless networks or portable 4G access points. Due to the limitations of those networks, some communications were not possible because our back-end servers could not access the devices at the demonstration locations with HTTP.

There are several ways to bypass the restrictions caused by NAT. The most well-known techniques are based on relay servers and hole punching [3]. For example, Lin *et al.* [4] discuss five NAT traversal techniques used by voice over IP applications. Different NAT traversal techniques also cause load on the gateway devices doing the work.

One option is MQTT [5] that is a common technology in the IoT domain. In this research, we wanted to see if MQTT can solve these problems and if our systems can be made MQTT compatible. Our initial studies also indicated that the changes to the original system would be minimal with MQTT. Thus, we wanted to port our system to use MQTT to ensure that our earlier research results can be applied under realistic network configurations.

2.2 Basics of MQTT

MQTT uses the publish-subscribe communication pattern [5]. 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 that manages the subscriptions and publishing of the messages. In MQTT, the subscriptions and published message are matched to each other with *topics* which may form hierarchical structures. Topics are constructed so that a slash character separates the different levels in the hierarchy. For example, if `abc` is the 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.

2.3 Research challenges

MQTT is not just a different protocol since it also imposes a different style of distribution. While RESTful applications built on top of HTTP are based on a synchronous request-response paradigm, MQTT is based on the publish-subscribe pattern. Change from request-response to publish-subscribe may potentially lead to big changes in the architecture of the system.

HTTP is a relatively heavy protocol both computationally and from network traffic perspective [6]. MQTT has lower power consumption than HTTP in many cases [7]. This is important in the IoT domain with low-resource devices. Our hypothesis was that MQTT uses less CPU, memory, and energy than HTTP. However, simple porting of a REST-based system on top of MQTT might not use MQTT efficiently; for instance, the number of communication may increase.

With the growing number of IoT devices, scalability needs to be taken into account in the IoT domain. For example, more devices mean more problems when devices in varying networks need to communicate with each other. Since the scale is expected to grow, the IoT domain should consider preparing for open IP networks to prevent problems coming from NAT traversal and mobility [8]. For example, some present IoT platforms support local connectivity but not global connectivity [9]. Since secure global connectivity and open IP networks are not available yet in the large scale, NAT traversal with MQTT takes part in helping with the scalability issues. In addition, when traversing NATs, security should not be forgotten.

3 Related work

3.1 Relation of MQTT to request-response and NAT traversal

Request-response over publish-subscribe architecture is not a new idea and different needs for such applications have been reported [10–12]. However, our case system and the need to overcome problems caused by the network gateways motivated us to investigate the topic again. A draft document by Advancing Open Standards for Information Society (OASIS) [13] acknowledges the lack of built-in request-response pattern in MQTT. They state that it is needed when (i) IoT device reads data from the server or from other device or vice versa, and (ii) when the IoT device needs to set a value in the server or other device or vice versa with a confirmation that the operation was successful. Our case adds a new third need – the logic of the application should be radically changed for MQTT without request-response pattern.

A few studies about the relation of MQTT and REST exist. Collina *et al.* [14] 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. Chen and Lin [15] implemented an MQTT proxy in their REST architecture comparing latency and performance between the protocols. However, they do not discuss how to implement functionality similar to REST with MQTT. Neither of these studies tries to overcome the network architecture restrictions that rise because of NATs or firewalls.

Bellavista and Zanni [16] also acknowledge the usefulness of hierarchical MQTT topics in modelling IoT device trees and MQTT's ability to help with NAT problems. Their aim is to use a combination of MQTT and constrained application protocol (CoAP) to get the best benefits from both since CoAP uses few resources in certain situations.

Uehara [17] presents a general framework for IoT devices where MQTT is used as a communication protocol. They use MQTT for communicating beyond NAT and for request-response traffic. Unfortunately, the details of request-response implementation are a bit unclear.

3.2 Tool tutorials

Some tool tutorials [18–21] describe solutions that use request-response with MQTT. While the techniques presented in the tutorials could be useful for tackling the restrictions imposed by the network architectures, we did not see enough benefits to integrate the presented complex tools or use different programming languages since our aim was in minimal refactoring. However, we briefly summarise their request-response techniques. Reactive Blocks [20] (a visual Java programming environment) and Emitter.io example [21] use request identifiers, unique request-response topics and subscription of hierarchical topic structures with wild cards. Eclipse Kura IoT service gateway [18] instructs to use response codes similar to HTTP codes and places REST verbs to MQTT topic hierarchies. The tutorial of solace systems [19] (a complex messaging middleware) has a different approach of using general reply topics and correlation IDs in message payloads for connecting replies to requests.

3.3 Alternative techniques

Techniques to implement the functionality without MQTT exist as well. HTTP long polling could work but we wanted to use MQTT because it supports IoT better. HTTP is not designed for pushing data from the server to client [7] and HTTP long polling is considered inefficient [22]. Websockets [23] could be another

alternative often used in web browsers to create two-way communication. However, WebSockets are not designed for constrained devices and they do not support IoT domain well [24]. The use of native Websocket library could still be one option.

Destination network address translation (DNAT) is an enhanced version of NAT that that allows remote peers to initiate a session with peers in the local network. It also improves network access and reliability issues [25]. To the best of our knowledge, the relation of DNAT and IoT has not been explicitly discussed in the scientific IoT literature. Gateways also need to be configured for DNAT which is not always possible. We needed a solution that works with any network and gateway.

Virtual private networks can be used for NAT traversal but it is not simple [26]. Universal plug and play have also been suggested for NAT traversal. However, it is often considered insecure (and thus disabled in gateway devices) and it is not supported by all NATs [27]. It is a relatively resource consuming technique to be used in constrained IoT devices [28] and it does not work well with nested NATs [29].

Srirama and Liyanage [30] propose a solution based on transmission control protocol hole punching [31] for mobile devices in 3G and 4G networks. The approach could also be applicable in the IoT domain. However, studies suggest that functionality of hole-punching depends on the NAT implementation [32]. Although the use of hole punching for this kind of IoT application calls for future research, in this research we decided to use MQTT instead.

Other IoT protocols than MQTT exist. For example, Open Mobile Alliance Lightweight Machine-to-Machine [33] is an IoT protocol that supports device and application management [34]. LWM2M uses CoAP [35] protocol. CoAP is a lightweight version of HTTP and it is assumed to have the same NAT problems as our original implementation. As with HTTP, port forwarding or particular connection requests can be used with CoAP [36] but several authors [16, 17, 22] mention the benefits of MQTT when communicating beyond NAT.

3.4 Summary

Better support for request-response pattern has been proposed to the next version (5.0) of MQTT by OASIS [37]. The solution combines three main ideas: ‘reply-to’ topics included in the request, correlation-ID to connect the requests to replies, and unique ‘reply-to’ topics, but unfortunately, the available material does not provide a detailed explanation about them yet. The built-in request-response seems promising but, in contrast, our solution works with MQTT 3.1.1.

As a summary, several other researchers have worked on the integration of HTTP/REST architectures to MQTT protocol. For example, many parties acknowledge the need of request-response in MQTT, MQTT is connected to HTTP systems in various ways and the benefits of MQTT when communications behind NAT are known. However, there seem to be relatively few scientific publications about using MQTT in request-response (or REST) style or design of MQTT topic hierarchies.

4 Proof of concept

4.1 REST-style request-response with MQTT

When a system that uses HTTP and follows RESTful architectural style is ported on top of MQTT, some problems need to be solved. In our example case, we recognised the following main challenges:

1. *Addressing of the resources*: While in HTTP, the addressing is uniform resource locator (URL)-based and in RESTful architectures, all resources have a unique URL, MQTT clients subscribe to topics.
2. *Request-response architecture*: In HTTP, all sent messages assume a response that includes information of about success or failure, and an optional payload. MQTT messages are one directional without a response.

The addressing problem is reasonable easy to solve since the naming conventions of URLs and MQTT topics are very similar. For example, a URL http://example.com/abc/123 can be mapped to a three-level topic `example.com/abc/123`.

There are a few alternatives to implement the responses. We need a mechanism to connect requests to corresponding replies. Also, the broker's ability to deliver messages efficiently should be taken into account. We reasoned a few options to implement responding. We give two examples in this study:

1. Before sending any request, the caller subscribes to a unique topic for the response to the request it will send next. 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 be designed so that the topic for the response message can be derived from the request automatically. Unsubscription from the response topic needs to be done so that the number of registered topics in a long-living system does not grow without a limit.
2. Each caller has a generic response topic that it subscribes to, and all responses to that caller are sent to that topic. The respondent needs to know the response topic, the response message needs to include request identification, 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 requester from the broker to the client. 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 with the previous option, a smaller number of topics is needed and they are not created and removed dynamically.

After analysing the options, we selected the first option since it requires the least modifications to our application code, it uses the broker more efficiently, and since the number 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.

In the second option, the clients would need to do more work by parsing the payload to connect the requests to responses. Besides, a general topic is considered as an anti-pattern in a high throughput environment [38].

The status code, e.g. 200 indicates success, is core component of the HTTP protocol. In principle, it could be encoded to the topic hierarchy or added to the payload. In our case, adding it to the payload requires few changes to the original source code, but also keeps the topic hierarchy simpler.

4.2 Original system

The present work is based on our earlier research on programmable IoT devices [1]. In that work, we have developed a system for development and deployment of applications to IoT devices. The original system consists of three active components: integrated development environment (IDE), runtime environment and resource registry (RR). The IDE runs on a 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 provides a representational state transfer (REST) API for installing, starting, stopping and removing applications [39]. For example, when a new application is installed on a device at address `example.iot`, the IDE sends a POST request to URL http://example.iot/APP and the payload of the request is the application package in TGZ format.

All devices and installed applications need to register themselves with a central registry RR. The RR also provides an API for the discovery of devices, device capabilities, installed applications and services provided by the applications. The architecture of the system is shown in Fig. 1. The arrows depict communication between the components: IDE deploys and manages applications in devices, devices register themselves and update their applications to RR, and IDE queries devices and

installed applications from RR. All the communication in the original system is implemented using HTTP. Further information about the original system can be found in [1, 40].

It should be noted that the logical control-flow of the original system requires a response from the previous request before sending the next. For example, IDE should not query the updated state of the applications (six in Fig. 1) from the RR before it gets the acknowledgment of a deployment (4 in Fig. 1) from the device.

4.3 MQTT implementation

We use node.js [41], MQTT.js [42], and Mosquitto [43] MQTT broker which implements MQTT version 3.1.1. We implemented the communication from the IDE to devices and from devices to RR seen in Fig. 1 with MQTT. From the communication shown in Fig. 1 the communication between IDE to RR is still implemented with HTTP. The resulting architecture and communication are presented in Fig. 2. The right side on Fig. 3 shows how an MQTT broker is accessed by the other parties behind NATs. 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 problem in implementing communication from IDE to RR with MQTT as well – if that is needed. Note that RR is a central component in the IoT deployment system whereas the MQTT broker only delivers MQTT messages.

From the different options [(1) complex topic and requestID in the topic, (2) general topic and requestID in message], to implement the request-response pattern with MQTT discussed in Section 4.1, we selected the one where a unique topic is created for every request and response. The topic hierarchies designed for our system are presented in Fig. 4. One notable detail in the hierarchies is the relation of replies to corresponding requests. In our solution, each topic mapped to a resource has request and reply subtopics followed by unique identifiers (rID). Device identifications (dID) are used as unique identifiers for devices.

The left hierarchy in Fig. 4 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 of topic hierarchies for messages that are directed to them. Essentially this implements the addressing scheme discussed in Section 4.1: 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 1 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 of 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. 4. The first level indicates that the hierarchy is meant for RR. The topics `RR/devices/request and RR/devices/reply` are used for registering devices. Device identification is not used yet because the unregistered device does not have a dID yet. On the second level, each registered device has a separate branch identified by dID. 'Apps' branch is used for managing the applications running on the device or retrieving information about them. For example, it is used by devices to publish the current states of all its applications to RR using topic (`RR/devices/<DID>/apps/request/<rID>`). As with device hierarchy, if an individual application resource needs to be expressed, the hierarchy can be extended by adding appIDs after 'apps' level.

The sequence charts in Fig. 5 describe how an HTTP request-response relates to MQTT request-response. One request-response sequence between device and RR requires multiple communications between three components, device, MQTT broker and RR. The message sequence chart in Fig. 5 depicts the devices' registration both with HTTP and MQTT. 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.

In Fig. 6, 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 to 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.

5 Evaluation

5.1 Amount of refactoring

One of our targets and success indicators was the number of required changes to the original system. We noticed that the changes were relatively small and local. The adaptation to MQTT was done by replacing the parts of the source code that sends the HTTP requests with a source code that publishes an MQTT message and subscribes to the reply topic.

The code examples in Fig. 7 show how the original JavaScript code snippet using HTTP is refactored for MQTT. The operation shown in the example is a device registering itself 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 a string and the response status and message body are not automatically parsed. An alternative and possibly a simpler way could be to include the status code in the topic hierarchy but we did not try that in practice.

While the snippets are an example from our case, the solution can be generalised so that for using MQTT in request-response style, dedicated request-response topics together 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 a normal publish-subscribe pattern when needed.

5.2 Resource consumption

We did not discover any performance issue while testing. The maximum size of an MQTT message is about 256 MB and the size of the messages in our system has been under 10 kB. The requester removes temporary topics after receiving the response which increases the amount of traffic a bit but prevents the system from growing memory usage continuously. Our solution uses complex topics though simple and short topic structures could use fewer resources [38]. On the other hand, a flexible topic structure is important when adding new features. Still, MQTT is IoT optimised and using it should help to save the resources when compared with HTTP.

We conducted experiments to compare the resource consumption of the protocols with the request-response pattern. We compared two applications. The first sent 1000 requests and received replies for those with MQTT. The experiment setting was such that Raspberry Pi (our constrained IoT device) worked as a requesting MQTT client, the MQTT broker located in the OpenStack virtual machine, and a modern laptop had the replying MQTT client. The latency from Raspberry Pi to MQTT broker was 38 ms (throughput 14 Mbps), and from the laptop to MQTT broker was 2 ms (throughput 262 Mbps). The request workload was 'GET

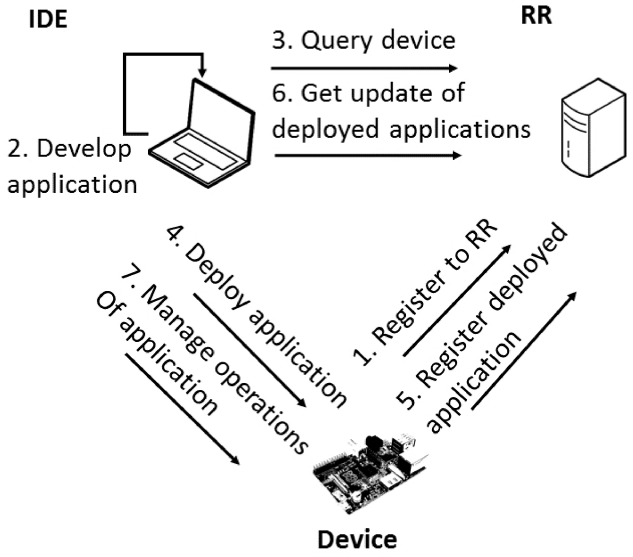

Fig. 1 *Original framework with HTTP. Numbers indicate an order of a typical workflow*

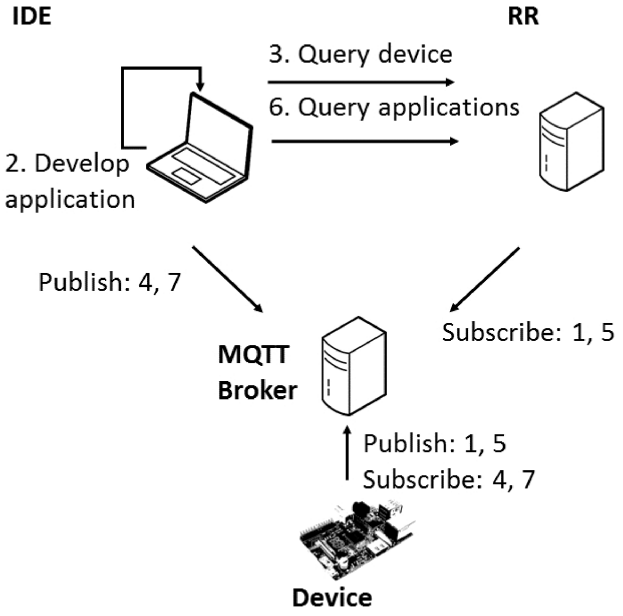

Fig. 2 *HTTP replaced with MQTT. Numbers correspond to the numbers seen in Fig. 1*

x' where x was a request ID integer between 0 and 999. The reply workload was '200 OK x' where x was the corresponding request ID. MQTT payload was symmetrically encrypted with AES-256. The second application did the same thing with HTTP without encryption. The setting was such that Raspberry Pi (again as constrained IoT device) worked requesting HTTP client sending requests to an Apache HTTP server. The latency from the Raspberry Pi to the HTTP server was 20 ms. The reply workload was empty.

The requests were sent as fast as possible without extra delay between them. The experiments were conducted on the Rasperry Pi version 2 (total memory 927 MB) running Raspbian operating system and Node.js [41] version 5.4.0. The used MQTT library was mqtt.js version 3.1.1. The MQTT broker was Mosquitto version 1.4.10.

The measurements were done using process.memoryUsage() function of Node.js and Linux command line tool *time*. MemoryUsage function shows the total memory allocation and time shows the used CPU time of a process execution.

Table 2 shows the results of five CPU time and memory consumption measurements. MQTT used on average 10.3 s less CPU time and HTTP used more than double the CPU time in every measurement. On average MQTT used 1.6% points less memory.

6 Discussion

The proof of concept works as expected and it 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. Refactoring of the system on the source code level required a relatively small amount of work. The MQTT implementation has a centralised broker that is accessible from everywhere. Thus, only the broker consumes a public IP address, and the devices can stay behind NATs and firewalls. As a result, our demonstration worked as expected.

Since the broker needs a public IP address, our proposal does not work if it is located in a private IP space nor if the firewalls have default MQTT ports blocked from inside to outside traffic. In some cases, it might be possible to tunnel MQTT to other ports than the default port but we did not study it.

Note that we do not think that adding request/<rID> or reply/<rID> to the topic addressing a resource (for example, `example.com/resource123/request/<rID>`) is a practice used in REST. Actually, such a solution might be even argued to resemble remote procedure call more than REST. Instead, we try to describe a way to use MQTT efficiently and conveniently while being consistent with REST as far as it is feasible. While the MQTT version of the REST method (for example GET) uses to request and reply topics, the target of the operation is identified by the base topic `example.com/resource123`.

Our experiments with 1000 request-replies suggest that the MQTT uses considerably less CPU time and less memory than HTTP even with a request-response pattern. While 1000 request-replies done as fast as possible might not be a real-world use case, it gives hint about the resource usage between the protocols. Reducing CPU and memory consumption improves energy-efficiency, which is getting increasingly important. The differences seen in our experiments can be significant on constrained devices even if you do not need to worry about NAT traversal.

In HTTP-based systems, the clients get either a response or an error condition. In the current MQTT system conditions like network errors may lead to situations where the 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 a 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 tests.

If extra security is needed in REST, there are standard mechanisms for encryption, authentication, and authorisation. MQTT also provides security features such as authentication and authorisation but those were not used with the proof of concept. While they were not used, we think using the following features would be relatively straightforward:

- Authentication by adding username and password while connecting to broker [36].
- Authorisation by using Mosquitto broker's topic permissions [44].
- Payload encryption [45]. (Our MQTT experiments done with symmetric payload encryption suggest that resource usage still remains lower than HTTP.)

In addition, transport layer security could (and should) be used but it can be challenging with constrained devices [45]. Secure MQTT [46] is also one proposed solution to improve MQTT security.

7 Conclusions and future work

We presented a work where originally REST-based communication was substituted with MQTT communication maintaining the request-response behaviour from the application point of view. The

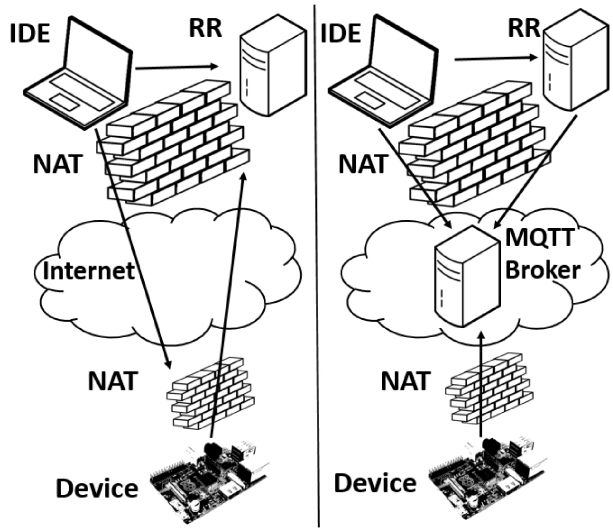

Fig. 3 *Left: NATs restricts communication with HTTP. Communicating with parties behind NATs is difficult. Right: MQTT broker in the public Internet allows communication despite the NATs. Similarly to left figure, the cloud presents the Internet*

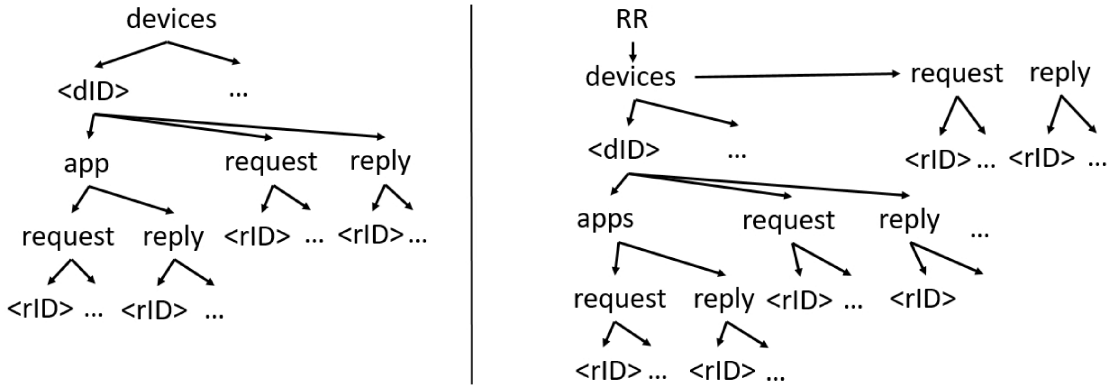

Fig. 4 *Topic hierarchies for MQTT implementation. Left: device. Right: RR*

Table 1 Mapping of the URLs to MQTT topics when a device sends requests to RR

URL	MQTT topic
RR-host-address/devices	RR/devices/request/<rID>
RR-host-address/devices/<DID>/apps/<APPID>	RR/devices/<DID> /apps/request/<rID>

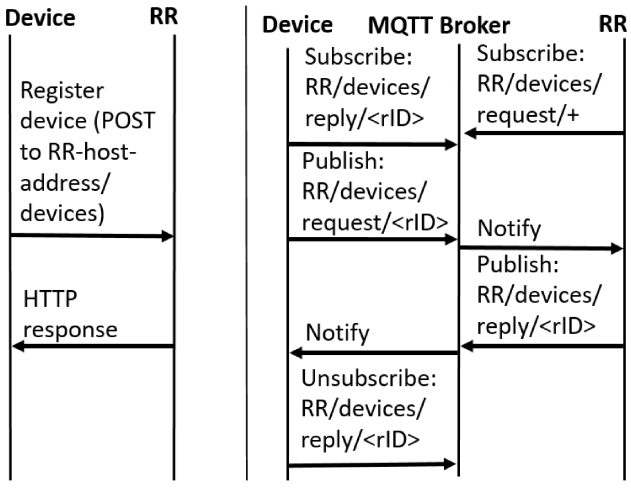

Fig. 5 *Left: register a device with HTTP. Right: register a device with MQTT*

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

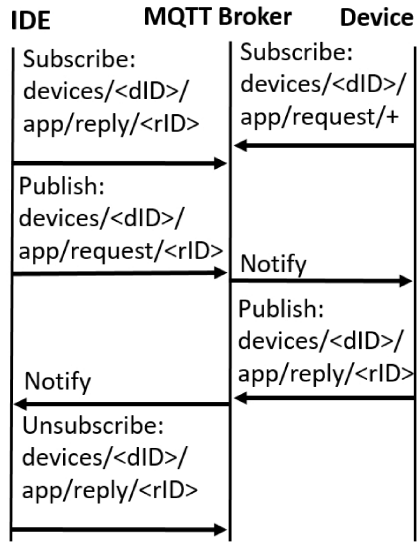

Fig. 6 *IDE deploys an application to a device*

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 a request-response paradigm. Our solution is based on separate response messages and design of topic hierarchy with specific request and response 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. Our comparison of resource usage between the protocols suggests that

```
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
    RRInfo.idFromRR
      body.toString();

    //Save dID to config file...
    // (removed from this snippet)
    } else {
    //Error handling..
  }
});
//-----------------------------------

//Subscribe for response and register
// the device. Use unique request ID.
client.on('connect', function ()  {
  client.subscribe('RR/devices/reply/' + rId);
  client.publish('RR/devices/request/' + rId,
    JSON.stringify(deviceInfo));
});

// Listen for responses and parse the
// status code from the body.
client.on('message',
  function (topic, message) {

  if(topic == 'RR/devices/reply/' + rId) {
    if(message.toString().substr(0, 3)
      == '200') {

        // Read dID from MQIT response.
        // It is after the status code.
        deviceInfo.idFromRR =
        message.toString().substr(5,
          message.length - 1);

        //Save dlD to config file...
        //(removed from this snippet)

        client.unsubscribe(
          'RR/devices/reply/' + rId);
    } else {
      //Error handling...
    }
  }
});
```

Fig. 7 *Top: Original HTTP version. Bottom: Original code refactored to use MQTT*

MQTT uses less CPU time and memory even with the request-response pattern, which is significant when using IoT devices with low resources.

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 has 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, which is not case in the real world. 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 in more detail in a real-world scenario to answer the questions rising from the growing IoT

Table 2 Means and confidence intervals of the measurements. Confidence level is 90%

	Mean	Confidence interval
HTTP CPU, s	17.72	[17.43, 18.01]
MQTT CPU, s	7.46	[7.37, 7.55]
HTTP memory, %	6.05	[6.04, 6.06]
MQTT memory, %	4.46	[4.45, 4.47]

phenomenon. Comparison with other approaches, like CoAP, HTTP long polling, and hole punching would also be an interesting research topic.

8 Acknowledgments

We would like to thank NOKIA for funding the work.

9 References

[1] Ahmadighohandizi, F., Systä, K.: 'Application development and deployment for IoT devices'. The Fourth Workshop on Cloud for IoT (CLIoT), Vienna, Austria, September 2016
[2] Datta, S.K., Bonnet, C., Nikaein, N.: 'An IoT gateway centric architecture to provide novel M2M services'. IEEE World Forum on Internet of Things (WF-IoT), Seoul, South Korea, March 2014, pp. 514–519
[3] Maier, D., Haase, O., Wäsch, J., *et al.*: 'NAT hole punching revisited'. 36th Conf. on Local Computer Networks (LCN), Bonn, Germany, October 2011, pp. 147–150
[4] Lin, Y.D., Tseng, C.C., Ho, C.Y., *et al.*: 'How NAT-compatible are VoIP applications?', *IEEE Commun. Mag.*, 2010, **48**, (12), pp. 58–65
[5] 'MQTT'. Available at http://mqtt.org/, accessed 5 October 2016
[6] Lampkin, V., Leong, W.T., Olivera, L., *et al.*: '*Building smarter planet solutions with MQTT and IBM websphere MQ telemetry*' (IBM Redbooks, 2012)
[7] 'Power profiling: HTTPS long polling vs. MQTT with SSL, on android'. Available at http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https, accessed 5 October 2016
[8] Naito, K.: 'A survey on the internet-of-things: standards, challenges and future prospects', *J. Inf. Process.*, 2017, **25**, pp. 23–31
[9] Kim, G., Kim, J., Lee, S.: 'An SDN based fully distributed NAT traversal scheme for IoT global connectivity'. IEEE Int. Conf. on Information and Communication Technology Convergence (ICTC), Jeju, South Korea, 2015, pp. 807–809
[10] Cugola, G., Migliavacca, M., Monguzzi, A.: 'On adding replies to publish-subscribe'. Proc. 2007 Inaugural Int. Conf. on Distributed Event-based Systems, Toronto, Ontario, Canada, June 2007, pp. 128–138
[11] Hill, J.C., Knight, J.C., Crickenberger, A.M., *et al.*: 'Publish and subscribe with reply'. (No. UVA-TR-CS-2002–32), Department of Computer Science, Virginia University, Charlottesville, 2004
[12] Rodríguez-Domínguez, C., Benghazi, K., Noguera, M., *et al.*: 'A communication model to integrate the request-response and the publish-subscribe paradigms into ubiquitous systems', *Sensors*, 2012, **12**, (6), pp. 7648–7668
[13] Technical Committee, O.A.S.I.S. M.Q.T.T. 'Request/reply message exchange patterns and MQTT version 1.0 working draft 02'. Available at https://www.oasis-open.org/committees/download.php/56280/reqreply-v1.0-wd02.docx, accessed 5 October 2016
[14] Collina, M., Corazza, G.E., Vanelli-Coralli, A.: 'Introducing the QEST broker: scaling the IoT by bridging MQTT and REST'. IEEE 23rd Int. Symp. on Personal indoor and mobile radio communications (PIMRC), Sydney, NSW, Australia, September 2012, pp. 36–41
[15] Chen, H.W., Lin, F.J.: 'Converging MQTT resources in ETSI standards based M2M platform'. 2014 IEEE Int. Conf. on the Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom), and IEEE Cyber, Physical and Social Computing (CPSCom), Taipei Taiwan, September 2014, pp. 292–295
[16] Bellavista, P., Zanni, A.: 'Towards better scalability for IoT-cloud interactions via combined exploitation of MQTT and CoAP'. IEEE 2nd Int. Forum on Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI), Bologna Italy, September 2016, pp. 1–6
[17] Uehara, M.: 'A case study on developing cloud of things devices'. IEEE Ninth Int. Conf. on Complex, Intelligent, and Software Intensive Systems (CISIS), Blumenau Brazil, July 2015, pp. 44–49
[18] 'Kura remote device management'. Available at https://eclipse.github.io/kura/ref/mqtt-namespace.html#mqtt-request/response-conversations, accessed 5 October 2016
[19] 'Request/Reply'. Available at http://dev.solacesystems.com/get-started/mqtt-tutorials/request-reply_mqtt/, accessed 5 October 2016
[20] 'Request/Response Pattern Over MQTT'. Available at http://www.bitreactive.com/mqtt-request-response/, accessed 10 May 2016
[21] 'Stock Explorer: Using Pub/Sub for Request/Response'. Available at https://www.codeproject.com/Articles/1159256/Stock-Explorer-Using-Pub-Sub-for-Request-Response, accessed 3 February 2017
[22] Fremantle, P.: 'A reference architecture for the internet of things'. WSO2 White Paper, 2014
[23] Fette, I.: 'The websocket protocol', RFC 6455, 2011
[24] Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F., *et al.*: 'A survey on application layer protocols for the internet of things', *Trans. IoT Cloud Comput.*, 2015, **3**, (1), pp. 11–17
[25] Wu, Z., Luo, H., Zhang, S., *et al.*: 'Design of a distributed network address translation system architecture'. Proc. Int. Conf. on Advanced Intelligence and Awareness Internet (AIAI 2011), Shenzhen, China, October 2011, pp. 207–210
[26] Kubota, A., Miyake, Y.: 'Public Key-based rendezvous infrastructure for secure and flexible private networking'. IEEE Int. Conf. on Communications (ICC'09), Dresden, Germany, 2009, pp. 1–6
[27] Eppinger, J.L.: 'TCP connections for P2P apps: a software approach to solving the NAT problem', Institute for Software Research, 2005, pp. 1–8
[28] Duquennoy, S., Grimaud, G., Vandewalle, J.J.: 'The Web of things: interconnecting devices with high usability and performance'. IEEE Int. Conf. on Embedded Software and Systems (ICESS'09), Zhejiang, China, 2009, pp. 323–330
[29] Müller, A., Carle, G., Klenk, A.: 'Behavior and classification of NAT devices and implications for NAT traversal', *IEEE Netw.*, 2008, **22**, (5), pp. 14–19
[30] Srirama, S.N., Liyanage, M.: 'TCP hole punching approach to address devices in mobile networks'. Int. Conf. on Future Internet of Things and Cloud (FiCloud), Barcelona, Spain, 2014, pp. 90–97
[31] Ford, B., Srisuresh, P., Kegel, D.: 'Peer-to-peer communication across network address translators'. USENIX Annual Technical Conf., General Track, Anaheim, CA, USA, 2005, pp. 179–192
[32] Pyylampi, T.: 'Design and implementation of a real-time multiplayer system for android operating systems'. Master Thesis (in Finnish), Tampere University of Technology, 2015
[33] 'OMA LightweightM2M V1.0 Overview'. Available at http://www.openmobilealliance.org/wp/overviews/lightweightm2m_overview.html, accessed 29 September 2017
[34] Rao, S., Chendanda, D., Deshpande, C., *et al.*: 'Implementing LWM2M in constrained IoT devices'. IEEE Conf. on Wireless Sensors (ICWiSe), Melaka Malaysia August 2015, pp. 52–57
[35] 'CoAP – RFC 7252 constrained application protocol'. Available at http://coap.technology/, accessed 29 September 2017
[36] 'MQTT and CoAP, IoT protocols'. Available at https://eclipse.org/community/eclipse_newsletter/2014/february/article2.php, accessed 3 February 2017
[37] OASIS MQTT Technical Committee.: 'MQTT request/reply MQTT-197'. Available at http://www.oasis-open.org/committees/download.php/58854/MQTT.Request-Reply.Overview-Sept.2016.pptx, accessed 16 August 2017
[38] 'MQTT essentials part 5: MQTT topics & best practices'. Available at http://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices, accessed 5 October 2016
[39] Ahmadighohandizi, F.: 'Liquid-IoT runtime environment (LIoTRE) API specification'. Available at http://farshadahmadi.github.io/, accessed 8 February 2018
[40] Hylli, O., Ruokonen, A., Mäkitalo, N., *et al.*: 'Orchestrating the internet of things dynamically'. Proc. 1st Int. Workshop on Mashups of Things and APIs (Middleware'16, 17th Int. Middleware Conf.), Italy Trento, 2016
[41] 'Node.js'. Available at https://nodejs.org/en, accessed 8 November 2016
[42] 'MQTT.js'. Available at https://www.npmjs.com/package/mqtt, accessed 5 October 2016
[43] 'Mosquitto – an open source MQTT v3.1/v3.1.1 broker'. Available at https://mosquitto.org/, accessed 5 October 2016
[44] 'Configuring and testing Mosquitto MQTT topic restrictions'. Available at http://www.steves-internet-guide.com/topic-restriction-mosquitto-configuration/, accessed 10 February 2018
[45] 'MQTT security fundamentals: MQTT payload encryption'. Available at https://www.hivemq.com/blog/mqtt-security-fundamentals-payload-encryption, accessed 10 February 2018
[46] Singh, M., Rajan, M.A., Shivraj, V.L., *et al.*: 'Secure MQTT for internet of things (IoT)'. Fifth Int. Conf. on Communication Systems and Network Technologies (CSNT), Gwalior, India, April 2015, pp. 746–751