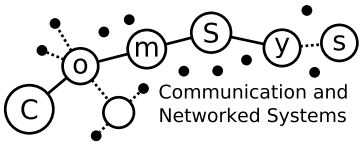
FACULTY OF
COMPUTER SCIENCE



Communication and Networked Systems

Master's Thesis

Software Updates for the Internet of Things: An Extension and Evaluation of the SUIT Implementation in RIOT

Vera Clemens

Matr. 223070

Supervisor: Prof. Dr. rer. nat. Mesut Güneş
Assisting Supervisor: M.Sc. Marian Buschsieweke

Institute for Intelligent Cooperating Systems, Otto von Guericke University Magdeburg

July 01, 2021

Abstract

Firmware updates are essential for the secure operation of Internet of Things (IoT) devices, as several recent examples of uncovered security flaws in IoT device firmware show. Firmware update mechanisms must fulfil several requirements. Firstly, as IoT devices are constrained in terms of available computing power, memory, battery power, and network bandwidth, firmware update processes must be designed with special consideration to these constraints. Secondly, firmware update mechanisms must be secure; if they are not, they allow an attacker to exhaust the device's resources through a Denial of Service (DoS) attack, block the installation of security patches, or even install arbitrary firmware. Thirdly, ease of use for the IoT device owner must be considered; ideally, it should be possible to install firmware updates without manual user intervention and they should cause minimal service downtime. The Software Updates for Internet of Things (SUIT) working group at the Internet Engineering Task Force (IETF) has published a draft for a standardized firmware update process that is designed to fulfil these requirements. SUIT has already been implemented for the IoT operating system RIOT using Constrained Application Protocol (CoAP) as the application layer protocol to transmit the update. In this work, we first analyse the requirements for secure software update mechanisms in the context of the IoT and provide a survey of existing mechanisms. We then design a new transport mechanism that uses MQTT For Sensor Networks (MQTT-SN) and extend RIOT's implementation with it. Finally, we evaluate and compare the transport mechanisms MQTT-SN and CoAP in a realistic testbed environment with respect to their resource requirements (flash memory, RAM and energy consumption), network protocol overhead, total update duration, and their scalability, i.e. ability to update multiple devices in parallel.

Contents

List of Figures

List of Tables

Listings

Acronyms

6LoWPAN IPv6 over Low-Power Wireless Personal Area Networks. 5–7, 26, 38, 40, 43, 45, 47, 50

AMQP Advanced Message Queueing Protocol. 11

CBOR Concise Binary Object Representation. 19, 26, 41

CoAP Constrained Application Protocol. v, 2, 3, 5–7, 10–14, 20, 21, 26, 30–33, 37–50, 59

COSE CBOR Object Signing and Encryption. 19, 26, 41

DDoS Distributed Denial of Service. 1

DDS Data Distribution Service. 11

DoS Denial of Service. v, 2

DTLS Datagram Transport Layer Security. 24, 34

HTTP Hypertext Transfer Protocol. 5–7, 10, 13, 23, 49

IANA Internet Assigned Numbers Authority. 34

IEEE Institute of Electrical and Electronics Engineers. 37

IETF Internet Engineering Task Force. v, 2, 17

IoT Internet of Things. v, 1–3, 5, 7, 8, 10–18, 20–24, 26, 28–30, 32–34, 37–39, 44, 46, 48–50, 59

IP Internet Protocol. 6

IPv6 Internet Protocol version 6. 5, 26, 38

LwM2M Lightweight Machine-to-Machine. 18, 26, 27

MAC Media Access Control. 47, 48

MAC Message Authentication Code. 19

MIoT Lab Magdeburg Internet of Things Lab. 37, 40, 50

MQTT Message Queuing Telemetry Transport. 2, 3, 5, 7–14, 23, 24, 27, 29, 30, 33, 34, 45, 46, 49, 59

MQTT-SN MQTT For Sensor Networks. v, 3, 5, 8–14, 29–35, 37–50, 59, 60

MTU Maximum Transmission Unit. 5

NIST National Institute of Standards and Technology. 2

OS Operating System. 3, 15, 22, 23, 38, 59

OTA Over-The-Air. 41

QoS Quality of Service. 7, 8, 10–14, 30

RAM Random Access Memory. 2, 3, 16, 17, 26, 30–32, 35, 37–41, 44, 47, 50

RSMB Really Small Message Broker. 33, 38, 45, 46, 59

RSSI Received Signal Strength Indication. 38, 45

RTT Round Trip Time. 11–14

SUIT Software Updates for Internet of Things. v, 2, 3, 11, 15, 17–29, 31–34, 37–39, 41, 42, 45, 47, 49, 50

TCP Transmission Control Protocol. 5–9, 12–14, 49

TDMA Time Division Multiple Access. 47

TUF The Update Framework. 16, 21, 22, 24–28, 49

UDP User Datagram Protocol. 5, 6, 8, 9, 12–14, 24, 26, 32, 38, 49

URI Uniform Resource Identifier. 6, 19–21, 34, 42

XMPP Extensible Messaging and Presence Protocol. 11

CHAPTER 1

Introduction

In contrast to the standard Internet, where humans are senders or receivers of information, the Internet of Things (IoT) consists of non-human "things", e.g. lightbulbs or fridges, that communicate with each other using Internet protocols. They are usually low-powered devices with relatively limited memory and computing capabilities. Since the expected lifetimes of those devices are of considerable length, it must be possible to update their firmware when bugs or security problems become known. Otherwise, the devices run the risk of malfunctioning or becoming compromised, especially if they are connected to the public Internet.

Several cases of security flaws in IoT devices that have become known in the past couple of years have illustrated why software updates for IoT devices are so important. For example, in 2016, security flaws were found in IoT surveillance cameras that had been sold in German and Swiss supermarkets. The cameras came with an insecure default configuration in which no password is set for the control panel, and automatically set up a port forwarding in the home router to make their video and audio streams publicly available over the Internet. A firmware update addressing the vulnerability had actually been available for months at the time of publication, however, over a third of scanned devices were still vulnerable [1]. In 2020, the so-called "Ripple20" vulnerabilties were found in a TCP/IP implementation which was used by hundreds of millions of embedded and IoT devices. Some of them allowed remote code execution. In response to this, an updated version of the implementation was released; however, it was difficult to identify all of the affected devices, and it was expected that some of the affected devices cannot or will not ever have their firmware updated [2]. An analysis of firmware images for over sixty different HP printer models found that over 80% of them contained third-party software with known vulnerabilities [3]. Finally, the Mirai botnet illustrates the magnitude of attacks that are possible using IoT devices. The botnet was involved in some of the largest known Distributed Denial of Service (DDoS) attacks ever in 2016. At its peak, it had infected approximately 600,000 IoT and embedded devices worldwide [4].

These examples make it obvious that software updates are an important part of the operation and maintenance phase in the lifecycle of an IoT device. The need for software updates for IoT devices is already widely acknowledged. At the time of writing, the most recent security guidelines published by the German Federal Office for Information Security

(German: “Bundesamt für Sicherheit in der Informationstechnik” (BSI)) and the American National Institute of Standards and Technology (NIST) both list the installation of firmware updates, especially security patches, as a basic requirement for the secure operation of IoT devices [5, pp. 3–4][6, p. 9]. The issue has even been acknowledged by European legislature: In May 2019, the European parliament passed a directive [7] which establishes a new right for consumers to receive necessary software updates for purchased goods with digital elements such as IoT devices. The German Federal Ministry of Justice and Consumer Protection has already published a draft bill implementing the directive [8], which assumes an average length of five years during which the manufacturer is required to provide these software updates, starting at the date of purchase.

However, there are several different problems with the current practices for IoT software updates. Some of them have been noted by Schneier [9], who points out that there are currently not enough incentives for IoT device vendors to provide software updates, and that there is no system through which device owners get notified of new software updates. Furthermore, ease of use and user acceptance must be considered. Currently, some device owners may lack the expertise required to install firmware updates on their devices. One survey found that in general, the average user is not likely to immediately install software updates when prompted, e.g. due to inconvenience or past negative experiences with buggy updates [10]. It must also be taken care that the software update process itself is secure and does not introduce new attack vectors, such as installation of a malicious software, rollbacks of security fixes, and Denial of Service (DoS) attacks that exhaust a device’s resources (e.g. battery power) by continually attempting illegitimate updates. For example, a vulnerability was found in the remote firmware update functionality used in HP LaserJet printers that allowed an attacker to install arbitrary firmware because updates were only authenticated using checksums and not signed [3]. The printers also did not require any administrator user authentication; anyone who was allowed to print could also install updates. Even if cryptographic signatures are used, attackers may compromise the signing keys. For example, Ronen et al. were able to extract the secret key used to encrypt and sign updates which was shared by many Philips Hue devices, and use it to sign and install malicious firmware [11]. If combined with another vulnerability in the ZigBee protocol, a worm could then spread itself from one infected light bulb to others within transmission range.

In answer to these problems, the working group Software Updates for Internet of Things (SUIT) at the Internet Engineering Task Force (IETF) is currently working on a standardized software update solution for IoT devices. The solution is intended to work even for constrained devices with as little as ~10 KiB Random Access Memory (RAM) and ~100 KiB flash memory (Class 1 devices [12]). Their efforts focus mainly on standardizing a format for the software’s metadata (also known as “manifest”), the software update architecture and the security mechanisms used to prevent attacks. They do not aim to standardize the mechanisms for transmitting the update to the devices, since IoT device operators are expected to already have certain infrastructure in place. Thus, SUIT aims to support all application layer (e.g. Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP)) and lower layer protocols (e.g. WiFi, IEEE 802.15.4) commonly used by IoT devices. It must be noted that the majority of the SUIT documents [13, 14, 15] are still in draft status and SUIT must thus be considered a work in progress.

The aim of this work is firstly to design and implement an extension of the SUIT implemen-

taton that is part of RIOT[1] [16], an Operating System (OS) for the IoT, which allows it to use another application layer protocol for update transmission. Prior to this work, only CoAP is supported. Secondly, the aim is to evaluate resource requirements, communication overhead and duration of the software update transmission in an IoT testbed. In summary, the contributions of this work are:

- Analysis and discussion of requirements for secure software update mechanisms in the context of the IoT
- Survey of other proposed update solutions and comparison with SUIT
- Design and implementation of a new transport mechanism for SUIT using MQTT For Sensor Networks (MQTT-SN)
- Evaluation of the implementation and comparison of different transport mechanisms (MQTT-SN and CoAP) in an IoT testbed under realistic conditions
  - Resource requirements (e.g. flash memory, RAM, energy consumption)
  - Network protocol overhead
  - Total duration of the firmware update
  - Scalability (updating multiple devices at once)

Thesis Structure The rest of this work is structured as follows: Chapter 2 provides background information on the IoT networking stack, especially the application layer protocols CoAP, MQTT, and MQTT-SN. Chapter 3 gives an overview over related works: It contains a survey of performance comparisons of CoAP, MQTT, and MQTT-SN. It also contains the discussion of the general requirements of a secure software update mechanism for IoT devices, a survey on other proposed software update mechanisms in the literature, and finally a comparison of the different ways in which these solutions and SUIT aim to fulfil the requirements. Chapter 4 describes the details of our implementation and also contains a discussion of design choices that were made. Chapter 5 describes the design of our evaluation and its results. Chapter 6 provides a summary and some pointers for future work on the topic.

[1] `https://riot-os.org/`

CHAPTER 2

Background: Application Layer Protocols for the Internet of Things

In this chapter, we provide background information on the application layer protocols used in this work: CoAP, MQTT and MQTT-SN.

Below the application layer, we use a widely used network stack for the IoT that consists of IEEE 802.15.4, Internet Protocol version 6 (IPv6) and User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) [17, 18]. Because IEEE 802.15.4 limits the frame length to 127 B, but IPv6 requires a minimum Maximum Transmission Unit (MTU) of 1280 B, an adaptation layer is required in between.

This is provided by IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN), which allows the transmission of IPv6 packets over IEEE 802.15.4 by fragmenting them [19]. It also provides compression for IPv6 and UDP headers to decrease overhead and prevent fragmentation for small payloads [20]. See Figure 2.1 for a visual overview.

2.1 Constrained Application Protocol (CoAP)

CoAP is an application layer protocol for the IoT used to retrieve, update or delete resources [21]. It was designed in the "request-response" style of the Hypertext Transfer Protocol (HTTP), and uses similar request methods and response codes, e.g. "GET" and

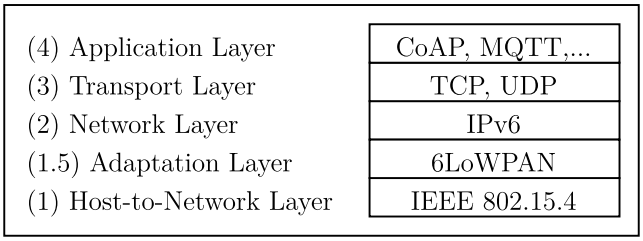
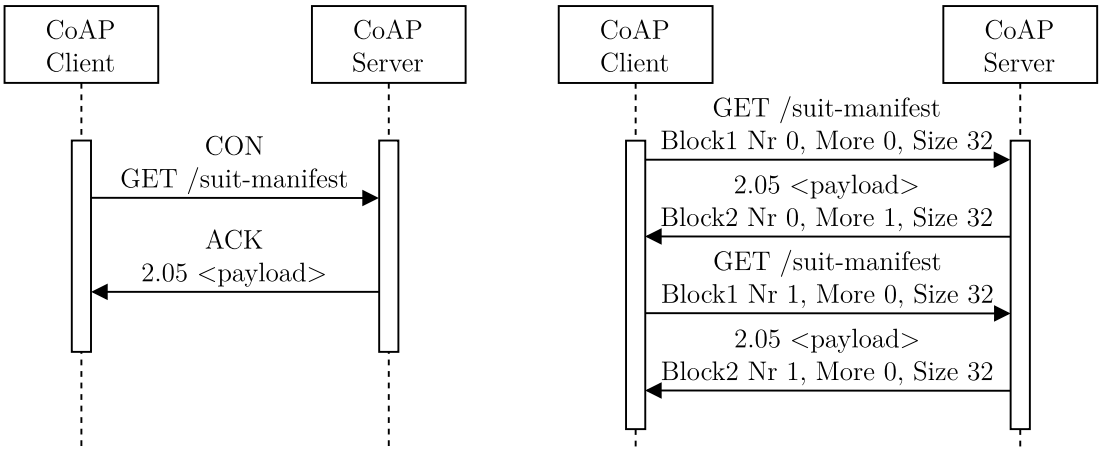


Figure 2.1: The IoT network stack used in this work.



a) Using confirmable messages: When the client sends a confirmable GET request for a resource to the server, the server must acknowledge the request. Here, it sends the requested resource payload in the same message as the acknowledgment.

b) Using block-wise transfer: The client requests a resource block by block. The server responds with the requested blocks using the requested block size. The final block is sent with the more bit unset.

Figure 2.2: Example CoAP requests and responses.

"404 (Not Found)". Also similar to HTTP, resources are identified by Uniform Resource Identifiers (URIs), which consist of the following parts:

```
coap[s]://<host>[:<port>]<path>[?<query>]
```

The special path prefix `/.well-known/` can be used to retrieve site-wide metadata, e.g. a list of all available resources. For an example CoAP request and response, see Figure 2.2a.

However, in contrast to HTTP, CoAP is more suitable for constrained devices. It does not require connection establishment prior to the exchange of requests and responses. The CoAP header can be as small as 4 B, while the HTTP header has a fixed length of 9 B [22, p. 12]. Additionally, CoAP's default transport layer protocol is UDP instead of TCP. This is beneficial for multiple reasons. Firstly, UDP is more lightweight than TCP because it offers fewer guarantees, e.g. messages are not acknowledged, retransmitted and may be delivered out of order. Secondly, UDP headers can be compressed by the 6LoWPAN adaptation layer (from 8 B to 2 B at maximum compression), while TCP headers (20 B) cannot [20, pp. 17–20]. Finally, TCP's congestion control mechanism is known to cause performance issues for wireless and mobile devices [23].

Due to the unreliability of UDP, CoAP itself implements an optional reliability feature using 2 B long message IDs that are contained in every CoAP message. Using this ID, duplicated messages can be discarded, and so-called "confirmable" messages can be acknowledged by the receiver. A CoAP message is marked as confirmable in the 2 bit "Type" header field. If a confirmable message is not acknowledged, it is periodically re-sent after timeout periods which exponentially increase in length until it is acknowledged or a maximum number of retransmissions is reached [21, pp. 21 f.].

CoAP messages should fit inside a single Internet Protocol (IP) packet to avoid fragmentation. Therefore, a maximum payload size of 1024 B is recommended [21, p. 25]. To

transmit larger amounts of data, block-wise transfer can be used [24]. In lossy networks, CoAP block-wise transfer is more efficient than 6LoWPAN fragmentation: When at least one of the 6LoWPAN fragments is still missing after a maximum wait time of 60 s, all other fragments are flushed and must be retransmitted [19, p. 13], while CoAP blocks are individually acknowledged and retransmitted. The block sizes of the request and response payloads are set separately using the "Block1" and "Block2" option, respectively. The block options each consist of three fields: a block number (4 bit to 20 bit), a bit indicating whether more blocks are following ("more bit"), and a block size (3 bit). Block sizes of $[2^4, 2^5, \ldots, 2^{10}]$ B are supported. When the block option is used by a client when requesting a resource, the server must respond using a block size no larger than the one requested. See Figure 2.2b for an example CoAP exchange using block-wise transfer.

CoAP also supports so-called cross-protocol proxies, which translate requests and responses between CoAP and another protocol. Because CoAP is quite similar to HTTP, translation between CoAP and HTTP can be easily done. This allows IoT devices that communicate over CoAP to access resources that are served by HTTP servers, and vice versa.

2.2 Message Queuing Telemetry Transport (MQTT)

The MQTT protocol [25] is based on the "publish/subscribe" paradigm, which means that devices can subscribe to so-called "topics" to receive data that they are interested in, e.g. `living-room/temperature`, as well as publish data on a topic. The subscriptions are managed by a "broker". When the broker receives published data, it checks which active clients are subscribed to the topic and forwards it to them. See Figure 2.3 for an example message exchange between a publisher, a subscriber and a broker. The size of the MQTT header depends on the message type, e.g. PUBLISH or SUBSCRIBE. The PUBLISH header has a minimum length of 3 B, plus the topic length.

MQTT extends the basic publish/subscribe paradigm with additional features, such as:

Quality of Service (QoS) Data can be published using three different QoS levels:

- Level 0 ("at most once") does not use any kind of acknowledgements (besides TCP acknowledgements between the client and the broker).
- Level 1 ("at least once") additionally uses PUBACK messages that are sent by the receiver (may be either client or broker) upon receiving the message. Duplicates of the message may still arrive after the PUBACK has been sent. They must be treated like new messages, because the message ID is free for reuse after the message has been acknowledged.
- Level 2 ("exactly once") uses a three-step process to ensure reliability and prevent duplicates: The receiver sends a PUBREC ("Publish Receive") message upon receiving the message, which lets the sender know to stop storing and retransmitting the message. The sender confirms this with a PUBREL ("Publish Release") message, which lets the receiver know it no longer needs to store the message ID to detect duplicates. The receiver finally replies with a PUBCOMP ("Publish Complete").

If PUBACK (QoS level 1) or PUBREC (QoS level 2) are missing, the message is retransmitted until it times out.

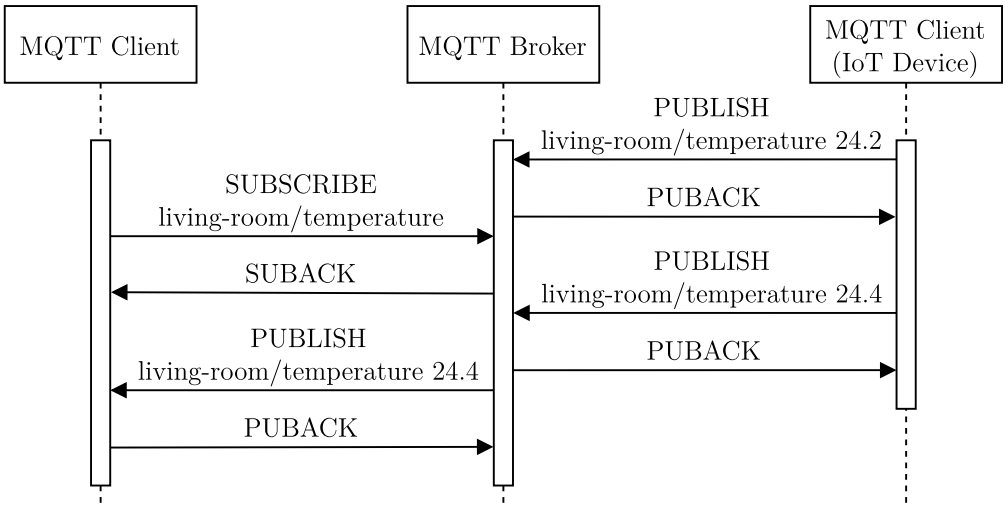


Figure 2.3: Example MQTT publish and subscribe messages: Clients can both subscribe and publish to topics. The broker forwards all published messages on a topic to the currently subscribed clients. The success of publish and subscribe actions is indicated using PUBACK and SUBACK messages when using a QoS level > 0.

Wildcard Subscriptions Two different wildcard symbols may be used to subscribe to multiple topics at once:

- The wildcard symbol `#` may only be placed on the final level of the subscribed topic, e.g. `living-room/#`. It matches the top-level topic and all subtopics, e.g. `living-room`, `living-room/temperature`, `living-room/light-status/lamp1` and `living-room/light-status/lamp2`.
- The wildcard symbol `+` may be placed at any level of the subscribed topic, e.g. `house/+/temperature`. It matches any single level, e.g. `house/living-room/temperature` and `house/dining-room/temperature`.

Retained Messages When the RETAIN bit is set in a PUBLISH message, the MQTT broker must store the message and send it to future subscribers of the topic upon subscription. It is useful when updates on a topic are irregular or infrequent because it ensures that any subscriber receives an initial value directly after subscribing.

There can be at most one retained message for a topic at any given time. If another message is published with the RETAIN bit set, it replaces the previous retained message.

If the QoS level of the retained message is set to 0, it may be discarded by the broker at any time; if the QoS level is 1 or 2, the semantics of the level must be honored, i.e. the retained message must be delivered at least once or exactly once to each future subscriber.

2.3 MQTT for Sensor Networks (MQTT-SN)

In the IoT context, one downside of the MQTT protocol is that it can only be operated using TCP on the transport layer, while UDP is generally preferred for constrained, mobile and wireless devices, as discussed in Section 2.1. Thus, a second standard called MQTT-SN [26] was developed specifically for these use cases, which uses UDP instead of TCP. It

Figure 2.4: Architecture components used by MQTT-SN: In addition to an MQTT broker, MQTT-SN requires a gateway that translates between its clients, which speak MQTT-SN over UDP, and the broker, which speaks MQTT over TCP. The gateway may be integrated with the broker or it may be a standalone component, as shown here.

supports all of the features described in Section 2.2 and the full set of message types defined by the MQTT protocol, except for AUTH messages, which are used for authentication using challenge/response methods.

In architecture, MQTT-SN can be considered an extension of MQTT. Besides an MQTT broker, it additionally requires an MQTT-SN gateway, which may be integrated with the broker or standalone. The standalone case is shown in Figure 2.4.

Besides the UDP support, MQTT-SN offers the following advantages for constrained devices and wireless networks:

Support for Topic IDs Instead of variable-length topic names, which can cause a large overhead on top of published messages, MQTT-SN uses 2 B long "topic IDs". Prior to use, they must be registered either by the gateway or by the client. Alternatively, MQTT-SN also supports pre-defined topic IDs which can be used without registration.

Topic IDs can also be used in version 5 of MQTT itself, where they are called "topic aliases". However, brokers are not required by the specification to keep using topic aliases when forwarding messages that were published using a topic alias. This is because publishers and subscribers are meant to be independent from one another, and thus a publisher's decision to use a topic alias cannot automatically carry over to subscribers. As a result, topic aliases are simply translated and never used for outgoing PUBLISH messages by some broker implementations such as the widely used Mosquitto.[1]

Server/Gateway Discovery Gateways may periodically broadcast ADVERTISE messages to allow clients to keep an updated list of online gateways. Additionally, clients may broadcast SEARCHGW messages to let others in the network know that they are looking for a gateway. Gateways or other clients may respond to this using a GWINFO message, which includes the address of an available gateway.

Keep-Alive for Sleeping Clients A client may let the gateway know that it is about to enter a sleep state by setting the "Sleep Duration" field in the DISCONNECT message. For that duration, the gateway will buffer packets on subscribed topics and deliver them when the client next comes back online and sends a PINGREQ to the gateway. After sending all buffered packets, the gateway answers the ping and the client may go back to sleep.

[1] `https://github.com/eclipse/mosquitto`

At the time of writing, the issue of using topic aliases for outgoing PUBLISH messages in Mosquitto is being discussed (see `https://github.com/eclipse/mosquitto/issues/1757` and `https://www.eclipse.org/lists/mosquitto-dev/msg02505.html`) and the feature is planned to be added in an upcoming version (see `https://github.com/eclipse/mosquitto/commit/de9780343b09d2d2d1c2bda6f2747c961e2fa2c1`).

A similar feature is also included in MQTT: If the "Clean Start" flag in the CONNECT message is set to zero, a persistent session is started. After it ends, the broker will continue buffering messages on the client's subscribed topics and deliver them when the client next connects. However, this is only mandatory for messages with QoS > 0, and the CONNECT procedure is less lightweight than the PINGREQ procedure.

QoS Level -1 Clients may publish messages without any previous connection setup with a gateway using the QoS level -1. This offers even less assurances than QoS level 0, because it is unclear whether the gateway adress is correct or the gateway is still online. However, this also saves the overhead of connection setup and teardown.

2.4 Comparison

In this section, we discuss advantages and disadvantages of CoAP, MQTT and MQTT-SN based on the protocol specifications which we briefly described in the previous sections. For an overview over performance comparisons of CoAP, MQTT and MQTT-SN implementations, see Section 3.1.

The publish/subscribe model used by MQTT and MQTT-SN has some advantages over the request/response model used by CoAP. As mentioned, MQTT and MQTT-SN can easily support sleeping devices by buffering packets at the broker. Additionally, the broker simplifies one-to-many or many-to-many communication between larger numbers of IoT devices, since they do not need to know each others network addresses to communicate, only the broker's network address. It must be noted that a publish/subscribe extension for CoAP exists [27], but it is still in draft status and thus has not been widely implemented.

Currently, another advantage of MQTT over CoAP is that it has a larger user base [28, pp. 38 f.]. There is no data on the number of users of MQTT-SN, but it can be assumed that it is the least used out of all three protocols. This is reflected in a much lower number of MQTT-SN implementations available, which also receive much lower contribution activity. For a short overview of available MQTT-SN implementations, see Section 4.2.1.

In summary, MQTT and MQTT-SN are most suited for the exchange of small amounts of current information such as sensor data in many-to-many fashion. MQTT-SN is more suitable for wirelessly connected constrained devices than MQTT, but it has a much smaller user base and lacks well-maintained software support. CoAP is most suited for the transfer of data of any size from servers to clients. Especially the transfer of large amounts of data is well-supported through the block-wise transfer option. It also provides interoperability with other protocols used in the wider Internet using cross-protocol proxies, most notably for HTTP.

CHAPTER 3

Related Work

In this chapter, we present and discuss related works. The first section presents an overview over previous comparative evaluations of the two application layer protocols CoAP and MQTT. The second section describes general requirements for secure firmware update solutions. The third section presents an overview over works describing firmware update mechanisms, including SUIT, and finally compares how they fulfill the requirements.

3.1 Comparative Evaluations of CoAP, MQTT and MQTT-SN

The performance of application layer protocols for the IoT has been evaluated and compared by several authors using several different metrics. This section gives an overview.

A similar survey of evaluations of application layer protocols, which includes not only CoAP and MQTT, but also Data Distribution Service (DDS), Advanced Message Queueing Protocol (AMQP) and Extensible Messaging and Presence Protocol (XMPP), was published by Dizdarević et al. [29] Unfortunately, MQTT-SN is not included in their survey.

It must be noted that not all performance differences measured can be ascribed to the protocols themselves. The specific implementation used obviously also has an impact on the measured performance. For example, a comparison of different CoAP implementations by Iglesias-Urkia et al. found significant differences in Round Trip Time (RTT), CPU and memory usage between them, even when they were implemented in the same programming language [30].

Transmission times

The message transmission time, delay or RTT is a well-studied metric that is included in most evaluations. Several of them find that MQTT performs better than CoAP, especially when packet losses and retransmissions occur:

Thantharate et al. measured the transmission duration of five consecutive messages in a simulated IoT testbed. Their results show that the average transmission time of MQTT messages with both QoS level 1 or 2 is at least three times as fast as that of CoAP confirmable messages. MQTT with QoS level 2 is slightly slower than QoS level 1 due to the increased acknowledgement overhead. However, it is unclear what causes the measured

time differences. By default, CoAP uses a stop-and-wait protocol which allows only one outstanding request or unacknowledged message at a time, while MQTT runs over TCP, which uses a sliding window protocol with larger window sizes. CoAP may have achieved better transmission durations if it was also configured to use larger window sizes.

Similarly, Bideh et al. measured transmission times and found that MQTT is up to seven times as fast as CoAP [31]. They state this is because TCP uses a more efficient retransmission algorithm which uses RTT measurements, while CoAP uses an arbitrary fixed value range (between `ACK_TIMEOUT` and `ACK_TIMEOUT` · `ACK_RANDOM_FACTOR`) for its acknowledgment wait timeout. Thus, TCP can trigger a first retransmission faster than CoAP when the network RTT is low. The effect can be lessened by decreasing the `ACK_TIMEOUT` parameter. However, even when decreasing it to the minimum of 1 s, they find that CoAP still performs worse than MQTT.

Collina et al. also found that MQTT performs with lower latencies when there is high packet loss [32]. They also state that this is because of TCP's faster retransmissions. Like Bideh et al., they were able to improve the performance of CoAP by lowering the parameters `ACK_TIMEOUT` and `ACK_RANDOM_FACTOR`. With these tuned parameters, CoAP achieves a performance similar to MQTT. However, the lowered parameters cause risk of unnecessary retransmissions, which may congest the network. RFC 7252 states that additional congestion control mechanisms must be used if `ACK_TIMEOUT` is decreased from its default value [21, p. 28].

Two evaluations consider the transmission of larger amounts of data. Bideh et al. measured transmission durations for a large file (870 KiB), which is transmitted in 870 blocks when using CoAP and in a single message when sent over MQTT [31]. In this case, MQTT is twice as fast. Mun et al. performed time measurements of the transmission of 50 kB large messages using different block sizes from 128 B to 1920 B and found that MQTT performs better for large block sizes of more than 1024 B because there is less fragmentation [33]. When sent over CoAP, these blocks are fragmented twice due to its limited payload size. These results show that fragmentation has a negative impact on transmission durations.

In some cases, CoAP performs similar to or even better than MQTT. In the evaluation by Mun et al., CoAP performs better than MQTT when messages are smaller than 1024 B [33]. Collina et al. find that when there is low packet loss, MQTT and CoAP perform the same [32]. Mijovic et al. also found that CoAP and MQTT with QoS level 0 achieve very similar RTTs, but MQTT with QoS 1 is slowed down by the additional application layer acknowledgements by a factor between 2 and 3 [34].

In their survey, Dizdarević et al. also found that the transport layer protocol is a major factor for performance, and that TCP-based protocols often achieve worse latencies in evaluations than the UDP-based ones [29]. Larmo et al. found in network simulations that MQTT messages arrive with increased delays when the TCP session is terminated in between messages due to TCP's three-way handshake [35]. They also found that if the IoT device's radio is periodically turned off to conserve energy ("duty cycling"), a minimum delay is added to each transmission originating at the device, especially if it is using TCP, because it cannot receive downlink messages such as the TCP Syn Ack of a connection establishment until it is awake again.

MQTT-SN is not included in many evaluations. Gündoğan et al. compared it against

CoAP [36] using ARM Cortex M3 nodes in the FIT IoT testbed [37]. They found that with a push/pull interval of 5 s, the push-based CoAP PUT and MQTT-SN PUBLISH do not differ significantly in the time required for the transmission, whereas the pull-based CoAP GET is slightly slower. With a shorter push/pull interval of 50 ns, the delays for confirmable CoAP PUT messages increase significantly due to application layer retransmissions. CoAP GET and MQTT-SN PUBLISH messages do not experience significantly decreased delays. In the evaluation by Mun et al., MQTT-SN performs the worst overall; however, they note that this may be due to the low level of maturity of the implementation used (Eclipse Paho) [33].

Energy consumption

Dizdarević et al. found that CoAP is more energy-efficient than MQTT in all surveyed comparisons, though both are more efficient than non-IoT-specific protocols like HTTP.

Larmo et al. also found in their network simulations that MQTT causes more energy consumption than CoAP due to the increased amount of radio transmission time caused by TCP overhead such as the three-way-handshake [35].

Bideh et al. transmitted payloads of different sizes and found that CoAP consumes slightly less energy for payloads of up to 1024 B since the overhead of connection setup is the dominant factor for such small payloads [31]. They conclude that data should be aggregated as much as possible to minimize energy consumption, since more frequent transmissions of small payloads are costly, especially when the communication setup and teardown need to be performed every time. For larger payloads, CoAP consumes significantly more energy than MQTT, again due to the fragmentation.

In the comparison of CoAP, MQTT and MQTT-SN done by Mun et al. [33], the energy usage measurements are highly similar to the total transmission times measured, which means that as described before, CoAP performs the best, except for large messages of more than 1024 B, where MQTT performs better due to less fragmentation, and MQTT-SN performs the worst overall. Mun et al. also find that the energy usage of all three protocols is heavily influenced by the network conditions (e.g. jitter and RTT): The worse the network conditions are, the more energy they consume.

Traffic overhead

Obviously, the traffic overhead incurred by headers and control traffic increases with the QoS level for MQTT. For CoAP, it increases when confirmable messages are used. In their survey, Dizdarević et al. found that CoAP consumes less bandwidth than MQTT in all surveyed comparisons. Our survey arrives at the same result. Unfortunately, MQTT-SN is not included in any overhead evaluations.

Chen et al. found that MQTT sends approximately 76 B of header and control data for every 100 B of payload, while CoAP sends only approximately 36 B for every 100 B of payload when using non-confirmable messages [38]. This increased overhead of MQTT is probably due to the control messages that are necessary before actually transmitting any payload, i.e. the CONNECT and CONNACK messages.

Mijovic et al. also found that CoAP has less protocol overhead than MQTT (both QoS 0 and 1), because the UDP header is smaller and there are no transport-layer acknowledgments [34]. When transmitting 100 B of payload, the payload makes up more than 30 % of

transmitted bytes when using CoAP, but less than 20 % of transmitted bytes when using MQTT.

Packet loss rates

Chen et al. found that MQTT experiences no packet loss even in lossy network conditions at the cost of increased latency for the received packets and increased control traffic overhead, while CoAP experiences packet loss rates approximately equal to the network packet loss rate [38]. This is not surprising, since lost MQTT messages will be retransmitted by TCP even when QoS level 0 is used, while the UDP-based CoAP does not retransmit packets by default. Unfortunately, they have not included CoAP's confirmable messages in their evaluation.

Gündoğan et al. also found in their measurements in the FIT IoT testbed that with a short transmission interval of 50 ms, significant packet loss occurs for non-confirmable CoAP messages, while confirmable CoAP and MQTT-SN PUBLISH messages did not experience packet loss because missing packets could be successfully retransmitted.

Conclusion

In conclusion, MQTT's largest downside when compared to CoAP appears to be the overhead caused by TCP due to the required connection setup and teardown and its larger header size. This overhead can lead to increased message transmission durations, energy consumption and traffic overhead, especially when the connection is re-established frequently. However, TCP also offers some advantage against CoAP, namely through its sliding window protocol compared to CoAP's stop-and-wait and its faster retransmissions. This gives MQTT an advantage especially in environments with high packet loss, where retransmissions are frequent. However, CoAP's approach is more suitable for constrained devices, because it does not require to keep RTT measurements or to support multiple simultaneously outstanding acknowledgments.

MQTT-SN is unfortunately not included in many evalutions. Gündoğan et al.'s results suggest that it offers a performance similar to CoAP. However, further evaluation is needed. As Mun et al. noted, the lacking maturity of implementations that are currently available may also negatively impact MQTT-SN's performance results.

Several evaluations note the negative performance impact of fragmentation. Here, CoAP is disadvantaged because it has a much lower maximum payload size than MQTT (1024 B vs. ~256 MB) and therefore forces more fragmentation at the application layer. However, CoAP's maximum payload size is much more realistic for constrained devices and networks and thus, in practice and for a fair comparison, MQTT should also be limited to a similar payload size.

3.2 Requirements for Software Update Mechanisms

In this section, we survey requirements that apply to all secure software update mechanisms.

First, we must clarify terminology, particularly the difference between the terms "firmware updates" and "software updates". "Firmware update" refers to the complete replacement of a device's entire software, while "software update" refers more generally to the replacement

of some part of a device's software with a newer version. It is less specific to the IoT or embedded context and also describes the update of a package on a Linux system, for example. Currently, most constrained IoT devices use full firmware updates only, however, it has been suggested that IoT software may become less monolithic in the future, so that different parts of it will need to be updated independently [39].[1] Smaller update sizes would also reduce network traffic and therefore the update duration and energy consumption [41, 42]. Thus, we use the more general term "software update" throughout this work where it is possible, as the SUIT standard drafts also do.

We now present a list of security requirements for software update mechanisms based on a survey of several other works. Firstly, the general information security principles of confidentiality, integrity, availability, and authenticity, which are named in the ISO 27000 standards series as the defining aspects "information security" [43, p. 4], apply to firmware update mechanisms:

Confidentiality It must be possible to encrypt both the software image and the metadata to ensure its confidentiality and prevent "read attacks". Attackers may use unencrypted images for reverse-engineering.

Integrity of the Image and Metadata The integrity of the software itself and its metadata must be verified. The device must check the integrity of the firmware update, i.e. that the update has not been tampered with along the way from the source to the IoT device.

Availability of the Updated Device An IoT device usually provides some service using its sensors and actuators. A software update usually requires a device to reboot and thus temporarily disrupts this service. It must be possible to schedule updates so that the availability of the service is not disrupted at critical times. The update process should also include fail-safes, e.g. for the case when a firmware update turns out to be invalid after it is received or the reboot fails [44].

Authentication and Authorization of the Update Source The update source must be authenticated and authorized. No unauthenticated or unauthorized source should be able to initiate a firmware update of a device. This can be done using cryptographic signatures, for example. Trust anchors or keys used by the device to perform the authentication may expire or change so that it is necessary to update them [45, 46]. Ideally, the device manufacturer or vendor should not be the only authorized source [3], so that the devices do not become obsolescent when they are no longer actively supported by the manufacturer or the manufacturer shuts down [47], which is undesirable both from a customer's and from an ecological perspective.

Samuel et al. define a set of principles that make a software update system resilient against attacks when a subset of signing keys has been compromised [48]. Firstly, there should be separation of responsibilities over multiple keys so that the attacker's power is limited when a key is compromised. Secondly, signatures using multiple keys should be required. Thirdly, it must be possible to revoke keys. Keys may also be automatically revoked after a certain amount of time or number of uses to encourage frequent replacement. Finally, keys used for very security-sensitive purposes should be stored on systems not connected

[1] For example, the embedded OS Tock (`https://www.tockos.org`) uses a non-monolothic approach. It separates the core OS from (third-party) applications that are dynamically loadable at runtime [40].

to the public Internet or not connected to a network at all. To prevent key compromises in the first place, widely trusted cryptographic algorithms and large key sizes should be used. They also identify the information that must be authenticated by a software update mechanism: the content of the updates, the availability of the updates, and the repository state (i.e. software versions available). They applied these principles in the design of The Update Framework (TUF), which is described in Section 3.3.2.

Secondly, there are security requirements related to the specific potential weaknesses of software update mechanisms. Some cases of such weaknesses were already described in Chapter 1. Cappos et al. classify several types of attacks on software repositories [49]: replay attacks (resending older versions), arbitrary package attacks (sending a different software than requested), freeze attacks (freezing the version a client sees and preventing updates)[2], extraneous dependency attacks (inserting additional dependencies that must be satisfied), and endless data attacks (sending endless amounts of data instead of the software image). These attack types can be carried out when an attacker compromises a software repository, i.e. is able to respond to client requests with arbitrary data, but they do not require a signing key to be compromised. Cappos et al. suggest that extraneous dependency attacks can be mitigated by signing update metadata which contains the dependency list, and replay attacks can be mitigated by including a timestamp in the software metadata that must not be older than the timestamp of the currently installed software. Endless data attacks can be mitigated by placing a cap on the maximum amount of data downloaded. To detect freeze attacks, they recommend to use signed root metadata files that contain hashes of all package metadata files served by the repository. The files are small, so they can have short expiry times and be updated and resigned frequently [49].

Karthik et al. describe further types of attacks [51]: read attacks (reading contents of firmware updates, e.g. to reverse-engineer), slow retrieval attack (slowing down transmission of firmware updates to exploit vulnerabilities in the meantime), and drop-request attacks (blocking network traffic of firmware updates).

In addition to replay attacks, an attacker may also send an outdated software version that is newer than the currently installed version, but still not the latest available one [15, p. 18]. Langiu et al. call this the "update freshness" property [52], which must be ensured.

Finally, there are also some usability requirements that should be met by a software update solution:

Suitability for Constrained IoT Devices An attempt at a definition of the term "constrained device" has been made in RFC 2778 [12]. According to this RFC, devices can be "constrained" in the sense of limited amount of flash memory, RAM, processing power, available (battery) power or accessibility once deployed. Additionally, the networks to which they are connected may themselves be constrained in some way, e.g. by low throughput or high packet loss rates.

The software update mechanism must adhere to these constraints. It cannot require memory or processing power exceeding the capabilities of constrained devices. It should have minimal energy consumption, which means that the network traffic and the total processing and transmission time spent on the update should be minimized [53, 42].

[2]This type of attack was also previously described by Bellissimo et al. [50]

Additionally, IoT devices are sometimes installed in inaccessible locations. Therefore, software updates must be possible over wireless connections [53, 42]. Devices may also not be constantly connected to a network if they are mobile. In these cases, pull-based approaches are preferable over push-based approaches [50].

Minimal User Interaction Software updates are expected to work automatically, i.e. with minimal or even no user involvement or attendance [13, 39, 9, 44]. Some types of IoT devices may not be able to ask for confirmation because they lack an appropriate user interface [50].

However, there are several reports of IoT devices losing certain functionalities or even being bricked after installing software updates [54, 55, 56]. Also, software updates may include "breaking" changes. A software update that changes the behaviour and output of an IoT device, e.g. the format of the sensor data it reports, may break systems that rely on this output down the line [45, p. 18]. Therefore, it must be possible to turn off automatic updates.

Targeting Subsets of Devices It should be possible to limit the installation of a firmware update to a certain subset of devices [42, 44]. Most obviously, a firmware should only be installed on the devices with the matching type of hardware. Additionally, updates may be targeted using other device attributes, such as location or group (e.g. test or production devices). Some use cases may additionally require that the update mechanism ensures atomicity, i.e. that either none or all of the targeted nodes successfully install an update [41, 42].

Monitoring It should be possible to monitor the state of the IoT devices, including at least their currently installed firmware version and whether an update was successful or not [44, 45].

Partial or Differential Updates As discussed before, an IoT device's software may be modular. In that case, the software update mechanism needs to support partial updates, which may be installed e.g. using dynamic linking. Also, differential updates may be desirable to minimize the network traffic [41, 45].

3.3 Software Update Mechanisms

In this section, we describe the design of several existing software update mechanisms, starting with SUIT. We focus primarily on update mechanisms designed for constrained devices. We specifically examine if and how they meet the requirements described in the previous section.

3.3.1 Software Updates for Internet of Things (SUIT)

The SUIT working group at the IETF aims to standardize a secure software update mechanism for constrained devices with as little as ~10 KiB RAM and ~100 KiB flash memory (Class 1 according to RFC 7228 [12]). This would lower development costs for IoT device vendors, who would no longer need to design and implement their own update mechanism, and ensure a certain level of quality. The SUIT documents specify a format for update manifests, the update architecture and the update process including the security mechanisms

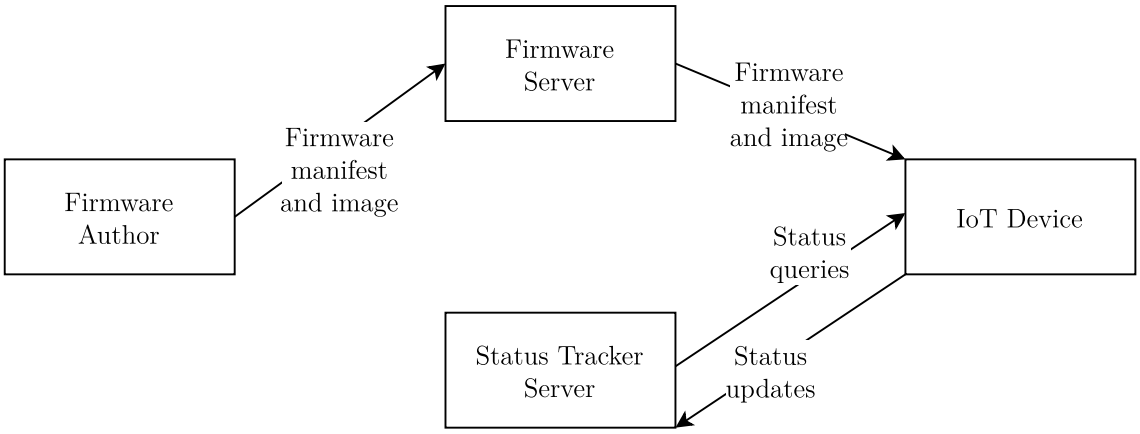


Figure 3.1: The components of SUIT's architecture: Besides the IoT device itself, a firmware and a status tracker server are required. The firmware server serves the software and manifests, while the status tracker server triggers and monitors updates.

used. Transport mechanisms, status tracker functionality and discovery mechanisms for status trackers and software repository servers are explicitly not standardized, because they may be implemented in various ways using already existing protocols, e.g. the Lightweight Machine-to-Machine (LwM2M) protocol for IoT device management. SUIT is not limited to full firmware updates. It supports the updating of arbitrary data, including partial software updates, configuration data, or cryptographic keys [13, p. 3].

Although SUIT is still a work in progress, there is significant interest in its early support, e.g. from the many researchers using RIOT [57], the open-source IoT operating system used in this work. An implementation of SUIT was added to the RIOT code base in October 2019[3]. SUIT was chosen for integration over other software update mechanisms because RIOT focuses on providing support for open, standard-based protocols and specifications.

Update architecture

The SUIT architecture consists of three components [13, pp. 7 f.]. It is shown in Figure 3.1. Firstly, there is the IoT device itself. It has one or more microcontrollers with associated bootloaders whose software may be updated. The device's software must implement update functionality which allows it to interact with the tracker and firmware servers, parse and verify the firmware manifest and store the downloaded firmware image. Secondly, there is the firmware server, which stores and distributes the firmware manifests and images. Finally, there is the status tracker server. It notifies devices of new firmware versions and receives status information from the devices (e.g. current firmware version, available flash memory). It can trigger a firmware update in a device. The firmware and the status tracker servers may be co-located on the same machine.

[3] `https://github.com/RIOT-OS/RIOT/pull/11818`

Update manifest format

SUIT manifests contain metadata about an update. Their structure is defined in the standard draft describing the manifest[4] [14], while the manifest fields are defined in the information model [15]. In total, the draft defines 24 manifest fields, seven of which are mandatory to include, such as a sequence number and a storage location. See Table 3.1 for an example manifest which was generated using the manifest generator tool that is included in RIOT.[5] The SUIT manifest also contains command sequences which describe how the receiver should process the manifest. Commands are either conditions that must be true or directives that must be executed. They are available for all processing steps usually required by an update process, i.e. signature and digest verification, parameter comparisons, fetching, copying and transformation of data, and code execution.

The manifests are encoded using Concise Binary Object Representation (CBOR), a binary data format specifically designed to generate small messages and require only little code to parse and generate [59], and are authenticated using CBOR Object Signing and Encryption (COSE) [60]. They are distributed in a container called the "SUIT Envelope" alongside an authentication wrapper containing one or more COSE signatures or Message Authentication Codes (MACs). When using the RIOT SUIT tool[6] to generate the signed version of a manifest, a single COSE signature of the hash of the manifest is added to the authentication wrapper.

Update process

The SUIT update process consists of the following steps [13, pp. 9–11]:

Table 3.1: An example SUIT manifest.

Field name	**Description**	**Example**
Manifest Version ID	Version of the manifest format used.	1
Manifest Sequence Number	Monotonically increasing sequence number, e.g. UTC timestamp.	1603813981
Vendor ID	Unique device vendor ID.	riot-os.org
Class ID	Class of devices that can accept the update.	nrf52dk
Image Size	Size of the update image in bytes.	85488
Offset	Storage offset in bytes.	8196
Image Digest	Hash digest of the update image.	`f5ff...`, SHA256
URI	Location from which the update image can be fetched.	coap://[2001:db8::1]/fw/nrf52dk/update-slot1.riot.bin

[4] At the time of writing, version 12 of the draft is the most current one, however the RIOT implementation of SUIT is still based on version 9 [58]. The changes to the manifest structure between the two versions are only minor, so that the description here applies equally to both.

[5] `https://github.com/RIOT-OS/RIOT/tree/master/dist/tools/suit/suit-manifest-generator`

[6] `https://github.com/RIOT-OS/RIOT/tree/master/dist/tools/suit/suit-manifest-generator/suit_tool`

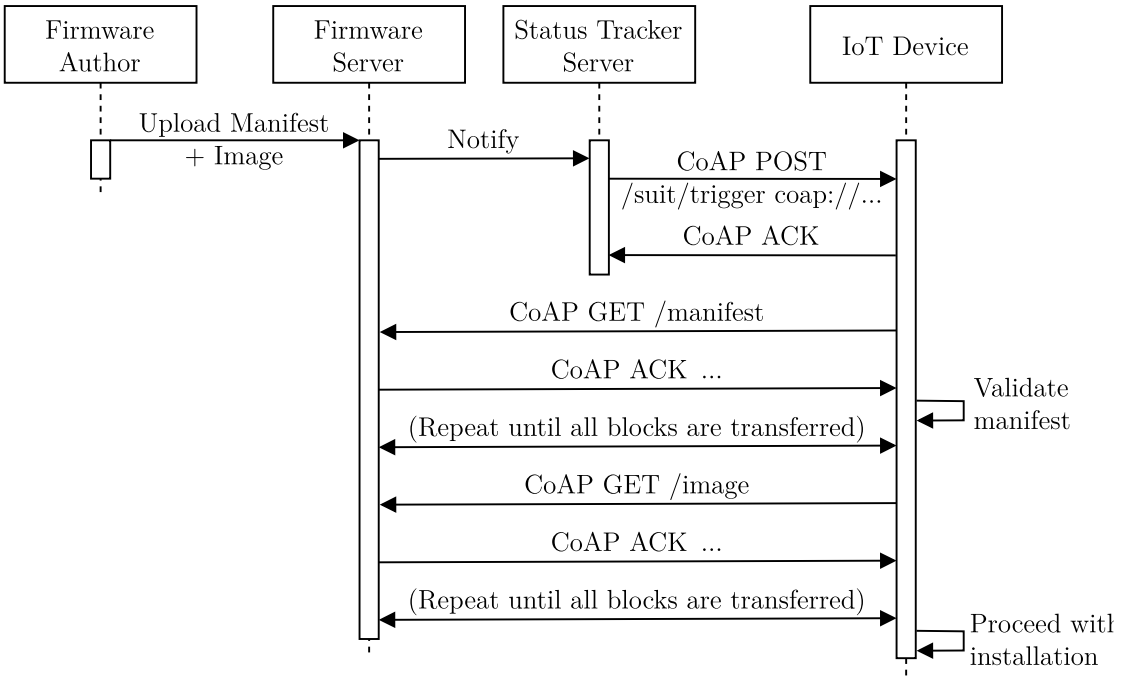


Figure 3.2: Sequence diagram of a software update using SUIT with a push notification and CoAP as the transport mechanism: The status tracker server pushes an update notification to the device using a PUT request. The device then downloads manifest and image using block-wise GET requests.

1. A new software update is uploaded to the firmware server by an authenticated and authorized source. The status tracker server is made aware of this.
2. The IoT device is notified of the newly available firmware image using a push-based approach, where the status tracker server pushes a notification to the device, or a pull-based approach, where the device periodically polls the status tracker server for available firmware images.

   The update manifest and image are transported from the firmware server to the device via an unspecified transport mechanism. Multicast or broadcast protocols may be used to efficiently distribute them to multiple devices at once.

   The device validates the firmware manifest's signature to authenticate the update's source. The manifest and the image can either be transported separately, or the image can be embedded into the manifest. As described above, the manifest contains the software's URI and an associated fetch command. If they are transported separately, the device can validate the manifest before starting the download of the image, and abort the update process if the manifest fails any check. The image is stored at the storage location specified in the manifest, e.g. at a certain flash memory offset.
3. Once the firmware image is downloaded, the installation may be initiated in one of the following ways:

   - The client immediately proceeds with the installation.

- The client delays the update until a certain condition is met (e.g. a certain time of day, battery level or until the device operator accepts [14, pp. 56–60]).
- The client waits for a trigger by the status tracker server.

The bootloader then boots from the new firmware image.

4. While the firmware update process is ongoing, the status tracker server tracks its progress, e.g. whether the update is in progress, successfully completed, or failed (and why).

See Figure 3.2 for the sequence of messages that are sent when CoAP is used as the transport mechanism. It shows how the current implementation of SUIT in RIOT works. As can be seen, a CoAP POST request is used by the status tracker server to inform the IoT device of the newly available update. The URI of the firmware manifest is transmitted within the payload of this request. The IoT device then pulls the manifest and the update itself from the firmware server using block-wise GET requests.

3.3.2 Others

In this section, we provide an overview over other proposed software update mechanisms, focusing primarily on mechanisms designed for constrained devices. Finally, we compare them to SUIT.

The Update Framework, Uptane and ASSURED

TUF is a framework for software update retrieval. It authenticates updates using cryptographic signatures and was designed to be able to survive the compromise of some of its signing keys using the four principles already described in Section 3.2. It was first described by Samuel et al [48], and is now a part of the Linux Foundation's Cloud Native Computing Foundation and in productive use at several companies.[7] The protocol specification is openly available [61]. TUF uses four different metadata files: `root.txt` (keys of root role and roles delegated by root), `release.txt` (latest versions, hashes and lengths of all metadata files except `timestamp.txt`), `timestamp.txt` (version, hash and length of latest `release.txt`), and `targets.txt` (available update files including hash digests, file size and optionally, custom information such as version or dependencies). Each metadata file has at least one associated role with a signing key that is only responsible for signing that file. Multiple signatures may be required. There can also be multiple hierarchical `targets.txt` files and keys, e.g. for update files provided from different sources. All metadata files also have an expiration time.

When downloading an update, the first step of TUF is to get the latest `release.txt` as specified by `timestamp.txt` and to check its length and hash. If `release.txt` indicates that a new `root.txt` is available, TUF starts over. If `release.txt` indicates that a new `targets.txt` is available, TUF updates its list of available files using the new version. When TUF is instructed by an upper software update layer to fetch one of the available files, it does so and checks its hash and length against the ones given by the `targets.txt` before returning the file.

[7] `https://theupdateframework.io`

Thus, the targets, release and timestamp roles share responsibility over the available update files so that all three must be compromised to allow an attacker to get a client to install malicious software. Freeze attacks are possible once the timestamp key is compromised because it allows an attacker to continually resign a frozen version of `timestamp.txt`. This prevents the release of a new `release.txt` and thus also of a new `targets.txt` or `root.txt`. However, the duration of the freeze attack is limited by the earliest expiration date of the other metadata files. Once the attack is noticed, the timestamp key can be replaced. Only a compromise of the root key(s) would be completely fatal to the system's security, because the attacker can sign a new `root.txt` containing his own keys for the other roles.

TUF was not designed for constrained devices, and it is not well-suited for them. It requires the updated device to choose which updates to install, to resolve dependencies and to validate multiple cryptographic signatures. Thus, update mechanisms based on TUF have been proposed that outsource these tasks to other components, e.g. ASSURED [62] and Uptane [51]. ASSURED extends TUF with a "controller" component which communicates with the repository on behalf of the constrained device, and takes on the tasks listed above and then transmits the update to the constrained device over a secure channel. The device only needs to confirm the authenticity of the controller and the constraints given in the update metadata, e.g. device class. Similarly, Uptane, which was created especially to update electronic control units (ECUs) in automobiles, adds a "director" role that is responsible for creating and signing a list of softwares to be installed specifically when an automobile requests updates. Due to the heterogeneous levels of constrainedness of the ECUs, one of them is chosen to be the primary who communicates directly with the director and performs the checks, and then distributes the software to all secondaries. Uptane's design also includes several other changes, such as an external time server which ECUs periodically contact with a nonce. The time server returns the signed current time and nonce. This is necessary because TUF requires the updated system to have an accurate clock to make use of the expiration timestamps. Uptane also uses an A/B slot system to ensure that there is always a bootable image available.

UpKit

With UpKit, Langiu et al. propose a software update mechanism that is supposed to be lightweight, efficient, open-source, and not dependent on any specific network protocol, hardware platform or OS [52]. Their goals therefore align quite closely with those of SUIT. Langiu et al. provide implementations for three different OSs (RIOT, Contiki-NG, and Zephyr) and multiple hardware platforms.[8]

UpKit's infrastructure consists of a vendor server, update server, and update client in the IoT device. The vendor server pushes new firmware images and signed metadata (the so-called "manifest") to the update server. The manifest contains the firmware version, size, hash, offset, and application and hardware platform ID. The update server announces the new version's availability. The update client can then send a request for it, including its device ID and a nonce. The update server appends the nonce to the update metadata, signs it and sends it to the update client. Using the nonce (if it is sufficiently random [63]), the freshness of the information is guaranteed. Unlike SUIT, they prefer nonces over timestamps because secure time sources are usually not available to IoT devices, and expiry timestamps

[8] `https://github.com/updatekit/upkit`

may be set too far in the future. Once the update client receives this information, it verifies both signatures and the metadata, such as the firmware version. If all checks are successful, the image is downloaded and its hash is compared. Finally, the image can be booted. UpKit supports the usage of flexible memory slots for update installation, e.g. A/B slots.

FiGaRo

Mottola et al. specify a software reconfiguration process for wireless sensor networks called FiGaRo (FIne Grained SoftwAre RecOnfiguration) [42]. It is implemented on top of the Contiki OS. In FiGaRo, the node software is structured into components that have interfaces (i.e. services they provide), dependencies on other components, and a version. Interfaces are represented as structs containing function pointers which always point to the implementation that is currently used. A management layer at the node keeps a representation of the current software configuration's dependency tree. When a new component arrives at the node, it checks whether all dependencies are satisfied. If they are, the component is installed and run if there is either no other component already implementing the same interface or the component that is currently installed has an older version. Components are replaced using dynamic linking. The component updates are limited to a subset of nodes using node attributes: First, each node's attributes must be declared (e.g. its hardware). They are piggybacked onto each outgoing message, overheard by all nodes in range, and used to build a mesh routing table that contains node attributes. Then, the target of the component update can be declared as a boolean function of the attributes (e.g. `Board == NUCLEO-F767ZI && Battery >= 75`).

Others

Frisch et al. present a software update mechanism for ESP8266-based IoT devices [44]. They use an infrastructure consisting of a firmware repository, a home automation controller and several IoT devices, which communicate over WiFi and MQTT. The repository serves firmware images and metadata files containing version numbers and cryptographic signatures of the images over HTTP. There is one metadata file per device type containing the metadata of the most current available firmware version. The devices use a two-slot system for storing firmware images. They publish state information like currently used firmware version and slot over MQTT after start-up. Periodically, they poll the repository for the newest version of the relevant metadata file. A poll can also be triggered by publishing a message on a certain MQTT topic. If a different version than the currently running one is returned, the download is started using an HTTP GET request.

While many firmware update mechanisms are available, none is yet well-established in the IoT. Schmidt et al. believe this is due to their overly complicated nature [64]. They therefore propose a new "minimal" mechanism. Their system consists of only a firmware server and the IoT devices. The firmware images sent by the server are encrypted using a pre-shared symmetric key and signed using the server's private key. They are accompanied by metadata (version, file size and IDs of keys to be used) which is also signed. The metadata is transmitted and validated first. As with SUIT, the transport mechanism used by the server is not specified. They implement a bootloader that is responsible for checking the validity of the signatures and the firmware version. Their system also uses a two-slot setup of the device's flash memory.

3.3.3 Comparison

In this section, we examine if and how SUIT and the other reviewed update mechanisms fulfil the requirements described in Section 3.2.

Confidentiality Schmidt et al. use symmetric keys to encrypt the transmitted firmware images, and include an encryption key ID in the update metadata. SUIT supports the optional encryption of the software image, also using symmetric keys. For encrypted software, the manifest includes an "encryption wrapper" that specifies which key should be used to decrypt the image and a list of processing steps necessary to decrypt the image [15, pp. 11 f., 15].

In any case, traffic may be encrypted using appropriate transport layer security protocols, such as Datagram Transport Layer Security (DTLS) for UDP.

Integrity of the Image and Metadata The mechanisms described by Frisch et al. and Schmidt et al. use signed images. As described above, TUF and its derivatives Uptane and ASSURED use signed hashes of the software images in their metadata files. Similarly, UpKit and SUIT include the hash of the image in the signed manifest.

When the IoT device receives the update metadata, it first validates the signature using the public key of an authorized entity (which must be already known to the device). During this validation, changes to the manifest by unauthorized entities can be detected. If the check is successful, the device compares the image digest contained in the metadata to the digest of the received image. In the case of SUIT, this check is part of the command sequence included in the manifest. During this check, changes to the image can be detected.

In any case, the integrity of the image and metadata can also be protected against man-in-the-middle attacks by transmitting it over a secured channel, as described above.

Availability of the Updated Device Uptane, UpKit and SUIT as well as the mechanisms described by Frisch et al. and Schmidt et al. support A/B updates using two memory slots, so that there is a firmware to fall back to if the reboot into the new version fails.

As described above, SUIT allows the scheduling of updates using commands that instruct the device to wait until a certain condition is met (e.g. a certain time of day or battery level). The mechanism described by Frisch et al. allows only rudimentary scheduling using update triggers published over MQTT.

An automatic rollback when the device cannot reestablish connection to the status tracker server after an update is not part of any of the surveyed mechanisms, but it could be added to the application logic.

Authentication and Authorization of the Update Source The authenticity of the update metadata is generally established through one or more signatures, which must be validated by the device. All mechanisms except FiGaRo and the one described by Frisch et al. use signed metadata. The mechanisms described by Frisch et al. and Schmidt et al. sign the update image directly.

In UpKit and SUIT, the authenticity of the update image is linked to the manifest's through the included hash digest. Furthermore, SUIT supports the usage of access control lists,

which must be implemented in application code, to grant different rights to different actors [15, p. 29].

Resilience Against Software Update-Specific Attacks See Table 3.2 for an overview of how SUIT defends itself against the different types of attacks introduced in Section 3.2.

Replay attacks using an outdated version are generally detected using the version number included in the update metadata. The mechanism described by Frisch et al. installs any version that differs from the currently installed one, even if it is older because they trust that an attacker will not gain access to their update release system. Uptane, UpKit, FiGaRo and SUIT as well as the mechanism described by Schmidt et al. require the version number to be newer than the currently installed version to mitigate replay attacks. Rollbacks are also possible, but they require the rerelease of an older version using a new version number. For this reason, SUIT calls it a "monotonic sequence number", not a version number, and suggests the usage of UNIX timestamps. TUF and its derivatives can also prevent the

Table 3.2: Mitigation strategies used by SUIT to defend against different types of attacks on the software update process.

Attack type	**Scenario**	**SUIT's defense**
Replay attack	Attacker sends a version older than the currently installed one.	Monotonic sequence number in the manifest (signed)
	Attacker replays an update meant for another device.	Vendor, device and device class ID conditions in the manifest (signed)
Arbitrary package attack	Attacker sends an arbitrary image instead of the requested one.	Image hash in the manifest (signed)
Update freshness attack	Attacker sends a version older than the most recent one available (may be newer than the currently installed one).	Expiration timestamp in the manifest (signed)
Freeze attack	Attacker serves outdated information and withholds updates without the device being aware.	Expiration timestamp in the manifest (signed)
Extraneous dependency attack	Attacker inserts additional dependencies to make the device download unwanted (malicious) data.	Dependency list in the manifest (signed)
Endless data attack	Attacker sends endless data instead of the requested manifest or image.	Image size in the manifest (signed)
Read attack	Attacker reads update traffic and may use the captured image for reverse-engineering.	Optional encryption

installation of an outdated version by removing it from the `targets.txt`.

Uptane, ASSURED, UpKit, FiGaRo and SUIT also include targeting information such as device class ID or serial number in the update metadata to mitigate replay attacks using updates meant for another device.

In special cases, an attacker may replay an update that is newer than the currently installed version, but older than the most recent one available. FiGaRo as well as the mechanisms described by Frisch et al. and Schmidt et al. will install this update. UpKit and SUIT are able to mitigate this attack using two different mechanisms, as described before. UpKit uses nonces that are generated by the device for a specific update request, while SUIT uses expiration timestamps in the manifest. Nonces have the downside of being difficult to generate with a sufficient level of randomness [63], while expiration timestamps require that a secure time source is available. Uptane includes a proposal for a secure time server suitable for embedded devices, however, it also uses nonces.

SUIT's optional expiration timestamp also allows a device to detect freeze attacks if all manifests that are still valid are republished with a new timestamp whenever they expire and the attacker has not compromised the signing key, similar to how TUF and its derivatives detect freeze attacks by the expiration of the metadata files.

Extraneous dependency attacks only affect the mechanisms which support dependencies, which are TUF, Uptane, ASSURED and SUIT. All of them mitigate against them by signing the dependency lists, which are included in the `targets.txt` and update manifest, respectively.

TUF, Uptane, ASSURED and SUIT and the mechanism described by Schmidt et al. can mitigate endless data attacks concerning the software image using the update file size which is included in the signed metadata. To mitigate endless data attacks on the metadata, a cap could be placed on the maximum metadata size that the device wants to receive, as suggested by Cappos et al. [49]. If the cap is exceeded, the device would abort the download.

Suitability for Constrained IoT Devices As described before, TUF is the only one of the surveyed mechanisms that is not especially suitable for constrained devices. All others are suitable in principle, but for all except SUIT, no comprehensive evaluation of the resource usage of the update process is available.

SUIT's implementation in RIOT has been evaluated by Zandberg et al. using three different off-the-shelf IoT devices with Arm Cortex M microcontrollers (M0+, M3, and M4), between 32 kB and 256 kB of RAM, and between 256 kB and 1 MB of flash memory [46]. They compare the memory usage (of RAM and flash memory) of a baseline configuration of a CoAP server without firmware update support to two different configurations with firmware update support using SUIT. The first configuration uses SUIT over the IoT network stack consisting of CoAP, UDP, IPv6/6LoWPAN and IEEE 802.15.4. The second configuration additionally uses the device management protocol LwM2M v1.0 on top of CoAP, which can be used e.g. to provision time information to devices. In total, the SUIT configuration requires approximately 121 kB of flash memory and 20 kB of RAM. The comparison shows this represents an overhead of approximately 16 kB of flash memory and 5 kB of RAM compared to the baseline configuration. However, the increase of memory requirements may not be as large for all use cases, because some of the components used by SUIT such as CBOR and COSE may already be part of some IoT applications. Additionally, if the

two-slot setup is used, flash memory is required for that also.

Furthermore, Zandberg et al. measure and analyse the total time spent on a full firmware update. They state that a majority of the total time is spent on network transfer (60 %) and signature verification (38 %), while all other tasks, e.g. parsing of the manifest, take only a negligible amount of time. Considering that approximately half of the firmware image's size is due to the cryptographic libraries that must be included for signature verification, they conclude that 68 % of the total update duration are caused by the signature verification. The choice of the cryptographic library used is therefore very important for the resulting firmware update speed.

Minimal User Interaction In all surveyed mechanisms, updates can proceed fully without user interaction. SUIT additionally supports optional user confirmation before a device proceeds with the installation of an update by including a user authorization condition in the manifest [14, p. 60].

Targeting Subsets of Devices In Uptane, the custom metadata fields can be used to limit updates to certain devices. When a vehicle polls the director for available updates, it includes its vehicle ID, which can be used to look up which ECUs are installed in the device. The director is then responsible for forwarding the compatible updates. In ASSURED, the process is very similar. The controller communicates with the software repository and performs the TUF validation steps on the received update files as well as checks local targeting rules. If all checks are valid, the update is forwarded to the appropriate devices.

As described before, FiGaRo supports very flexible targeting using fully custom node attributes, e.g. device class.

SUIT also allows the specification of custom conditions [14, p. 60]. Updates can be limited to subsets of devices by including conditions in the manifest, e.g. that the vendor ID, device class ID or device ID of the device that receives the manifest must match a given ID [14, p. 58]. Similarly, UpKit supports targeting using the application ID and hardware platform ID fields in the update manifest.

Monitoring In the mechanism described by Frisch et al., devices publish their state information such as the currently installed version on a well-known MQTT topic after a reboot.

Monitoring is a central part of SUIT in the form of the status tracker server. However, SUIT does not define specifically how and what should be monitored. This decision is left to the implementer. Existing protocols such as LwM2M may be used.

Similarly, in ASSURED, the controller receives a status update from the device once the update is complete.

Partial or Differential Updates Partial updates are a central feature of FiGaRo, as described before. Software is divided into components, which can be updated separately using dynamic linking.

SUIT supports both partial and differential updates. For differential updates, a list of required image versions must be included in the manifest to ensure that the correct base version is present before downloading the differential update. The hash digest of the base image must also be included in the manifest of the differential update and it must be checked using a condition before installation [15, p. 10].

Conclusion

When comparing how the surveyed update mechanisms fulfil the requirements listed in Section 3.2, some common features can be identified. All of them include metadata such as the version, file size and dependency lists with the update image. All except FiGaRo, which focuses on enabling partial updates rather than security, use cryptographic signatures to authenticate and check the integrity of update images and metadata. All except FiGaRo and TUF, which is not designed for constrained devices, support a two-slot system for storing firmware update images to memory while retaining the previously installed version as a fallback.

The design of TUF, which requires the usage of at least four different signing keys, differs the most from SUIT. TUF is similarly comprehensive and includes mitigation strategies for a variety of attacks, but it is not suitable for constrained devices. Of TUF's derivatives, Uptane is highly tailored to the automotive use case, while ASSURED is applicable to the general IoT use case. The main difference between ASSURED and SUIT is that in ASSURED, the controller component interfaces with the software repository on behalf of the IoT device, while in SUIT, the device itself interfaces with the firmware server. In ASSURED, the controller makes the decision which updates to install when. When the controller pushes an update to the device, it is already considered to be approved and the device is expected to install it immediately after performing basic integrity and authenticity checks. In SUIT, the conditions under which an update should be fetched and installed are specified in the manifest and the device itself evaluates them to determine how it should proceed.

The design of UpKit is most similar to SUIT. However, SUIT defines a more general and flexible metadata format than UpKit with optional support for dependencies, partial updates and update encryption, for example. Additionally, SUIT requires the validation of only a single signature, while UpKit always requires the validation of two signatures. This is a significant difference because the transmission and validation of signatures is expensive, as the evaluation by Zandberg et al. showed.

Overall, SUIT is a comprehensive framework for software updates for constrained devices. When compared to other update mechanisms, it does not lack any features. The manifest format, especially the command sequences that specify all steps of the fetching, validation and installation process as well as the conditions for installation, makes SUIT flexible, so that it can be used in various different use cases.

CHAPTER 4

Thesis Contribution: A New Transport Mechanism for SUIT Using MQTT-SN

In this chapter, we present the main contribution of this thesis, which is the design and implementation of MQTT-SN as a new transport mechanism for SUIT in RIOT.

4.1 Design

In this section, we discuss the considerations and decisions made during the design phase.

4.1.1 Choice of Application Layer Protocol

Firstly, we must decide which application layer protocol to chose for our new transport mechanism. We consider MQTT and MQTT-SN as options. As discussed in Section 2.4, both of them offer advantages: MQTT-SN is more suitable for wirelessly connected constrained devices and is therefore first choice for our IoT use case. Its major limiting factor is that it is currently not as widely used and well-supported as MQTT. For Linux systems, there are only very few MQTT-SN server implementations available, none of which are widely used and most are not actively maintained. However, on RIOT, there are actually two suitable MQTT-SN client implementations available. This may be due to RIOT's popularity with researchers, who are interested in MQTT-SN due to its advantages over MQTT, even if it is not widely used in practice. Thus, we choose to implement MQTT-SN as a new transport mechanism for SUIT.

4.1.2 Design of the Transport Mechanism

When designing the new transport mechanism using MQTT-SN, we must consider SUIT's distinct requirements. They differ from the most common use case for MQTT-SN, i.e. the transfer of sensor data, in three ways:

Firstly, the size of the data units that must be transferred during a firmware update (i.e. the manifest and the image) is much larger than the size of a sensor data unit (e.g. a

temperature reading). Unlike a sensor data unit, the firmware update data cannot be transferred in a single PUBLISH message. The message's size would exceed the size of the receive buffer in the IoT device's RAM. Therefore, the update data must be transferred in blocks; however, there is no feature like CoAP's block-wise transfer built into MQTT or MQTT-SN. Therefore, it must somehow be implemented by the application itself.

Secondly, the duration for which an update manifest and image are valid is much longer than the timespan during which a sensor data unit is valid. While sensor data readings may become outdated and be replaced by an updated reading quite frequently, e.g. multiple times per hour, the update manifest and image may be replaced by a newer version much less frequently. Still, the update manifest and image should only need to be published to the broker once when they first become available. Afterwards, the broker should keep them in storage and behave similarly to a file server.

Thirdly, the update manifest and image remain valid even if a newer version is available, whereas an older sensor reading is generally made obsolete by a newer reading. The broker will no longer serve the older sensor reading to publishers. To support software downgrades, the broker should however still serve older update images.

To fulfil these requirements, we use the RETAIN flag when publishing the update data (see Section 2.2). This causes the MQTT broker to store it and send it to all future subscribers immediately upon subscription. This way, IoT devices can request a download of the manifest and image by subscribing to the corresponding topics (see Figure 4.1). Since there can only be one retained message on a topic at a time, we must publish each version and each block on a separate topic. To separate images compiled for different device classes, we may also include the device class ID in the topic name, e.g. `suit/images/v2.0/f767zi/0`. To ensure a long-term storage, we additionally set the QoS level to 1, since a retained message with QoS level 0 may be deleted by the broker at any time. Using QoS level 2 is not necessary since the device can keep track of which blocks it has already received, and discard duplicate blocks. QoS level 2 would only increase the control overhead unnecessarily by tripling the number of acknowledgements per block.

To reduce the number of subscriptions necessary, wildcard subscriptions could be used. For example, the IoT device may subscribe to the wildcard topic `suit/images/v2.0/f767zi/#` (which includes the subtopics ending in `/0`, `/1`, `/2`,...) to receive all retained block messages. However, this would not lower the amount of control traffic. A topic ID must be registered for each of the subtopics. Without a wildcard subscription, the topic IDs are included in the SUBACKS. With a wildcard subscription, a REGISTER and REGACK message must be sent for each subtopic. Since the sizes of these control message types are very similar, the amount of control traffic is basically the same in both cases. Additionally, we must also consider the issue of message order. According to the MQTT protocol specification, the broker must forward messages published with QoS level > 0 to subscribers in the order that they were published in [25, pp. 97-98]. However, it does not specify in what order the broker should forward messages published on topics matching a wildcard subscriptions.[1] Therefore, the implementation cannot assume that the blocks arrive in any kind of order when wildcard subscriptions are used. This poses a problem because typical flash memory

[1] At the time of writing, the Mosquitto broker also uses publish-order for these messages (with the exception of the parent topic), however, this may change at any time. See: `https://www.eclipse.org/lists/mosquitto-dev/msg02463.html`

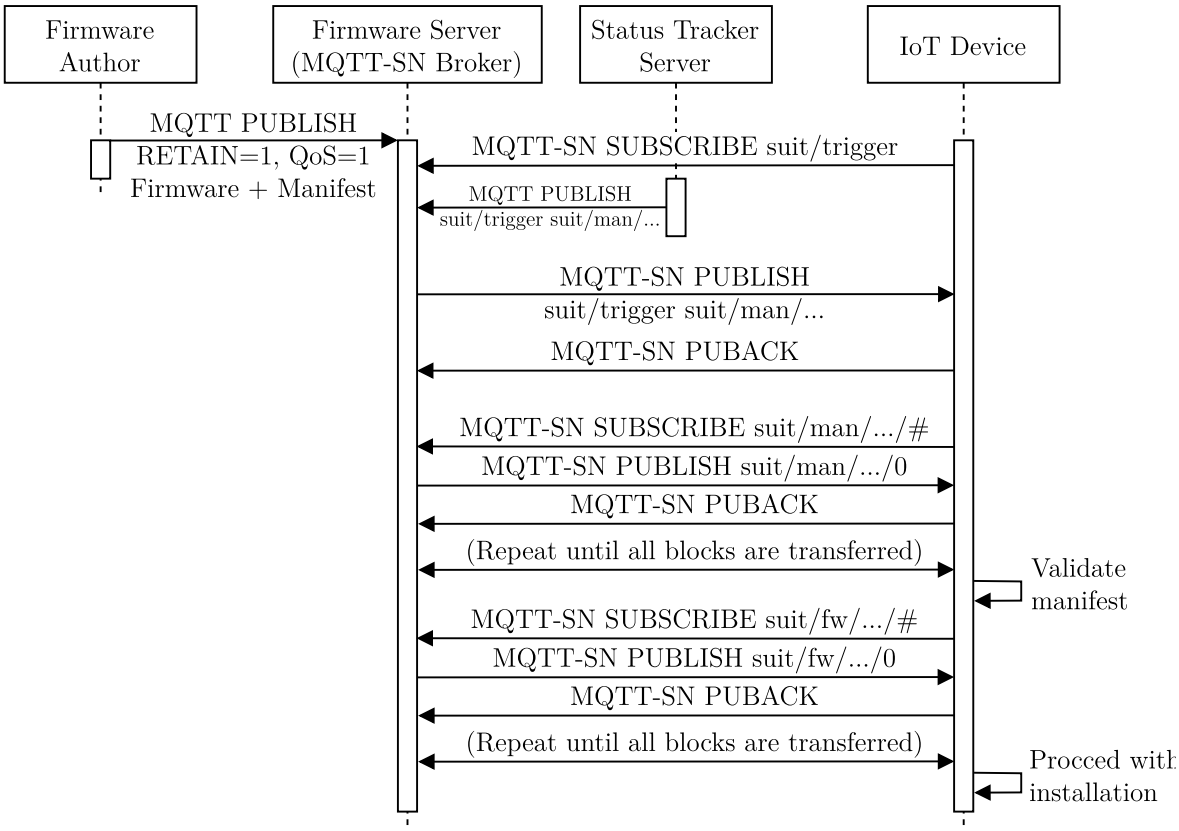


Figure 4.1: Sequence diagram of a software update using SUIT with a push notification and MQTT-SN as the transport mechanism: The status tracker server pushes an update trigger to the device using a PUBLISH message. The device then downloads manifest and image by subscribing to the block topics one by one and receiving each block as a retained PUBLISH message.

hardware interfaces first set all bits during erase operations and then only unset certain bits during write operations. This makes writing to flash memory in random order impractical, unless all of the writes are multiples of the page size in length as well as page-aligned. Thus, RIOT's APIs only support in-order write access to the flash memory. Out-of-order blocks would need to be buffered in RAM before they can be written to flash memory. However, in the worst case, the blocks arrive in reverse order, which would require almost the entire firmware image to be buffered in RAM. This is clearly not feasible. Additionally, neither of the MQTT-SN client implementations in RIOT actually support wildcard subscriptions at the time of writing. Therefore, we decide against the usage of wildcard subscriptions.

It must be noted that our design using retained messages has one major disadvantage when compared to CoAP: We cannot support variable block sizes as easily, because the block size must be fixed prior to the initial publishing. If multiple block sizes are required, the update manifest and image must be uploaded multiple times using different block sizes. However, software updates are only expected to work on devices of the same class, e.g. devices that have identical hardware. Thus, multiple different block sizes may not be needed, since a single block size can already provide a good match for the capacities of all targeted devices.

4.2 Implementation

To run SUIT on a RIOT device, the modules `riotboot` and `suit` are required. The module `riotboot` is a bootloader that partitions the device's flash memory into three sections: The first section contains the bootloader itself, and the second and third sections are firmware slots. Thus, two firmwares can be stored in flash memory at the same time. One slot is active, i.e. it is used during device boot-up. The inactive slot is used to store the newly downloaded firmware image during the update process. At the start of each firmware slot, the `riotboot` header is stored. It contains four 4 B long values: a magic number ("RIOT" in ASCII), the firmware version version, the start address of the actual firmware, and a checksum of the header.[2] The module `suit` contains the SUIT-related functionality, e.g. the different transport mechanisms. Currently, there is only one transport mechanism that is implemented, which uses CoAP. In this section, we describe the implementation of the new transport mechanism using MQTT-SN.

4.2.1 Choice of MQTT-SN Implementations

Firstly, we discuss our choices of MQTT-SN implementations both on the client and on the server side.

Client Implementation

As mentioned in Section 4.1.1, RIOT offers two different MQTT-SN client implementations: `emcute` and `asymcute`. The main difference between them is that `emcute` offers synchronous (blocking) functions, while `asymcute` offers asynchronous (non-blocking) functions. For example, consider how the two libraries implement the SUBSCRIBE request: The function `emcute_sub` returns only after the request is completed, either due to a server response or a timeout. In contrast, the function `asymcute_subscribe` returns as soon as the request has been sent, and a user-provided callback function is called when the response is received or the request has timed out. Internally, `asymcute` uses asynchronous UDP sockets that use an event loop to handle incoming messages and timeouts[3].

Its asynchronous design allows `asymcute` to support multiple simultaneously outstanding requests, e.g. PUBLISHes or SUBSCRIBEs. Each outstanding request is associated with its own request context data structure including a transmit buffer. In contrast, `emcute` uses only one global transmit buffer. For applications which make heavy use of requests originating at the IoT device, this can be a major performance advantage. However, these performance gains come with increased memory usage. A comparison of the memory usage of the two implementations shows that `asymcute` uses around 3.2 kB more flash memory and around 0.2 kB more RAM than `emcute`.[4] The increased memory usage is due to a larger code size caused by the library functions and the asynchronous sockets and slightly larger data structures used.

[2] `https://github.com/RIOT-OS/RIOT/blob/010dbcc9d3ff2890b3da2cb8926cb6ef7df2874d/bootloaders/riotboot/doc.txt`

[3] `https://riot-os.org/api/group__net__sock__async__event.html`

[4] `https://github.com/vera/masters-thesis/blob/master/results/asymcute_emcute_comparison/2021-02-19-memory-usage-asymcute-vs-emcute.txt`

In our firmware update application, the device frequently sends SUBSCRIBE requests (one for each block topic, see Figure 4.1), but since the blocks must be received in-order, each SUBSCRIBE must be completed before the next one can be sent. To allow the device to send multiple SUBSCRIBEs at once, potential out-of-order blocks would need to be handled, which would complicate the transport logic and would require an additional memory buffer. Therefore, `asymcute` does not appear to promise better performance than `emcute` that would justify the increased memory usage, which makes `emcute` the better choice for our application.

Server Implementation

When choosing the MQTT-SN server implementation used, we consider the following two options: the Really Small Message Broker (RSMB)[5] and the Eclipse Paho MQTT-SN Gateway implementation[6]. Both are open-source projects at the Eclipse Foundation.

The RSMB was developed at IBM and initially released as closed-source. The widely known Mosquitto broker was developed as an open-source alternative to it. The RSMB was eventually contributed to Eclipse as open-source in 2013, but it is no longer being actively developed and can now be considered deprecated. In its feature set, it is largely inferior to Mosquitto, which has a large user base and is being actively maintained. The only major advantage of RSMB is its MQTT-SN support, which Mosquitto and basically all other MQTT brokers lack.

The Paho MQTT-SN Gateway is a standalone gateway that translates between MQTT-SN and MQTT, which means that it must be used in combination with an MQTT broker (as seen in Figure 2.4). This is a major advantage over the RSMB, because it allows us to combine it with an up-to-date broker implementation such as Mosquitto.

Therefore, we initially tested the Paho MQTT-SN Gateway and the Mosquitto MQTT broker in combination. However, the Paho Gateway appeared to be lacking in maturity and suffered from several bugs (see Appendix A.2.1). It also achieved much worse total update durations in preliminary tests than the RSMB. Therefore, we decide to use the RSMB to obtain the best possible evaluation results.

4.2.2 Implementation of the Transport Mechanism

To add the transport mechanism using MQTT-SN to RIOT's SUIT implementation, we firstly add a tool that allows easy publishing of new firmware images and manifests. For the CoAP transport mechanism, the `make` target `suit/publish` does this by simply copying the manifest and firmware files to the folder that is being served by `aiocoap-fileserver`[7]. This makes the files available for retrieval using CoAP GET. For the MQTT-SN transport mechanism, we implement a Python program for the same purpose. The program receives a file path, block size and topic prefix as input parameters (among others) and performs a block-wise publishing of the file to the subtopics `<PREFIX>/0`, `<PREFIX>/1`, `<PREFIX>/2`, and so on. It additionally publishes the total number of blocks to the parent topic `PREFIX`.

[5] `https://github.com/eclipse/mosquitto.rsmb`
[6] `https://github.com/eclipse/paho.mqtt-sn.embedded-c/tree/master/MQTTSNGateway`
[7] `https://aiocoap.readthedocs.io/en/latest/module/aiocoap.cli.fileserver.html`

This information is used by the IoT device to know how many blocks to expect. It can also be used by the publishing program to delete previous retained blocks. This may be necessary when a file is published under an already used topic prefix but the number of blocks has decreased, either because the new file is smaller or because a larger block size is used. If the excess blocks were not deleted, they would be needlessly transmitted to all subscribing IoT devices from then on.

Secondly, we implement the actual transport mechanism. As shown in Figure 4.1, it involves mainly the handling of incoming PUBLISH messages on the trigger, update manifest and image topics. For the initial connection setup, we add two new shell commands: `con <ipv6 addr> [port]` for connecting to an MQTT-SN gateway and `sub <topic name>` for subscribing to a trigger topic. Ideally, a connection to an available gateway should be automatically established using gateway discovery (see Section 2.3), but this is not supported by `emcute`. It would also require some additionally security mechanisms such as the usage of DTLS to authenticate the gateways. If the gateway discovery process is not secured, any malicious server could announce itself using GWINFO messages and start serving invalid manifests or firmware images to execute resource exhaustion attacks. For monitoring, the device automatically publishes messages on three device status topics `suit/version/<DEVICE_ID>` (for the current firmware version), and `suit/slot/[in]active/<DEVICE_ID>` (for the indices of the currently active and inactive firmware slots) once it is connected to a gateway. For the handling of incoming PUBLISH messages, we implement three different handler functions:

Firstly, there is the handler for PUBLISHes on the trigger topic. These messages contain a manifest topic prefix. The device subscribes to the topic name contained in the payload and then to its subtopics `/0`, `/1`, `/2`,... to receive the number of manifest blocks as well as the blocks themselves, as described in Section 4.1.2.

Secondly, there is the handler for PUBLISHes on the manifest topic. These messages contain either the total number of blocks to be expected or a block of the manifest, as described above. The manifest blocks are copied to a 640 B buffer, which is large enough to hold an entire manifest, with an offset depending on the block number. If the final block has been transmitted, `suit_parse` is called on the manifest, which starts the validation of the manifest as described in Section 3.3.1, and if all checks are successful, the command sequences are executed, which usually contain a "fetch" command for the update image.

We extend the SUIT fetch command handler to handle manifest URIs beginning with `mqtt://` followed by a topic name[8] to call the fetch function defined by the MQTT-SN transport. This fetch function subscribes to the given topic and registers the third handler, which is for PUBLISHes on the firmware topic. Each firmware block is written to flash memory. If fewer bytes or more bytes than the total image size from the manifest are received, the fetch command aborts with an error. Otherwise, further commands (e.g. image hash check) are executed. If successful, the device finally reboots from the new firmware.

[8]This is not an official URI scheme. Currently, no URI schemes for MQTT or MQTT-SN have been officially registered at the Internet Assigned Numbers Authority (IANA). A proposal has been discussed by the MQTT Technical Committee (`https://issues.oasis-open.org/browse/MQTT-203`), however, it does not include topic names, which we require here. Therefore, we use this simple unofficial scheme.

```
1 typedef struct {
2   uint16_t num_blocks_total;
3   uint16_t num_blocks_rcvd;
4   uint16_t current_block_num;
5   uint16_t current_block_len;
6 } suit_mqtt_sn_blockwise_t;
```

Listing 4.1: The data structure used to keep track of the current state of the block-wise transfer over MQTT-SN.

A `struct` keeps track of the current state of the block-wise transfer (see Listing 4.1). The total number of expected blocks is necessary to know when the final block has been received. The number of already correctly received blocks is used to check whether the current received block is in-order, which is necessary because the firmware image must be written to flash memory without gaps, as described above. The number and length of the current block are used to calculate the correct offset and number of bytes to write when writing to both flash memory and the manifest buffer in RAM.

The source code of the implementation can be found at `https://github.com/vera/RIOT/tree/suit/mqtt-sn`.

CHAPTER 5

Thesis Outcome: Evaluation and Comparison

In this chapter, we present the evaluation of our implementation. The goal of the evaluation is to find out how well the new implementation of the MQTT-SN transport mechanism for SUIT in RIOT performs, especially in comparison to the existing CoAP transport mechanism. For this purpose, we conduct measurements of both variants.

5.1 Setup

In this section, we describe the parameters of our evaluation, such as hardware and software setup used, as well as the metrics that we examined. The workload which we evaluate is a full firmware update from start to end as outlined in Section 3.3.1, i.e. from the update trigger followed by transmission of the manifest and the update image until the successful reboot. The full firmware of the IoT device is updated.

We consider four metrics in our evaluation. Firstly, we measure both the RAM and flash memory requirements of our implementation. Secondly, we measure the total traffic volume going over the wireless link during the update. Since we do not vary the protocols used other than on the application layer (MQTT-SN or CoAP), we can evaluate which protocol causes more overhead. Thirdly, we measure the total time required for the full firmware update. Finally, we measure the energy consumption of a full firmware update.

For the time and energy consumption measurements, we use the Magdeburg Internet of Things Lab (MIoT Lab) [65], which is a testbed for IoT applications. At the time of writing, it consists of eight testbed nodes, which are positioned in several rooms on two levels of the computer science faculty's building at the Otto von Guericke University Magdeburg. The testbed nodes each consist of two commercially available hardware components: an x86 and an embedded device (Nucleo F767ZI). See Table 5.1 for an overview of the hardware specifications. The Nucleo boards are flashed and controlled by the x86 components, which are connected to the testbed management system via Ethernet. The boards are equipped with multiple transceivers, e.g. for Institute of Electrical and Electronics Engineers (IEEE) 802.15.4, IEEE 802.11n (Wi-Fi) and sub-GHz communication, and are also connected to the x86 components via Ethernet. The testbed nodes are capable of monitoring

the power usage of the board and the transceivers separately. The testbed uses the DES-Cript experiment description language to store experiment settings to ensure repeatability and reproducibility.

Table 5.1: Parameters that influence the results of the evaluation measurements. The parameters that are varied during the evaluation are highlighted.

Parameter	**Value(s)**
Hardware	
Boards	Nucleo F767ZI
. . . CPU	ARM Cortex M7 (STM32) @ 216 MHz
. . . Flash memory	2 MiB
. . . RAM	512 KiB
. . . Transceiver	Atmel AT86RF215
. . . . . Transmit frequency	2.4 GHz
. . . . . Transmit power	maximum[1]
Power monitoring	Espressif ESP32 + TI INA3221
MQTT-SN/CoAP server	Virtual 4 core CPU @ 2.4 GHz, 8 GB RAM
Testbed conditions	
RSSI between nodes	77.27 dBm
Other network loads	low[2]
Other loads on the MQTT-SN/CoAP server	none
Software	
IoT OS	RIOT
Network stack	GNRC
. . . Host-to-network layer	{Ethernet, IEEE 802.15.4}
. . . Network layer	IPv6 + 6LoWPAN
. . . Transport layer	UDP
. . . Application layer	{MQTT-SN, CoAP}
MQTT-SN server and client	RSMB and `emcute`
CoAP server and client	`aiocoap-fileserver` and `nanocoap`
SUIT configuration parameters	
Size of manifest	~500 B
Size of firmware image	~85 kB
Block size	{32 B, 64 B, 128 B, 256 B, 512 B}
Number of updated devices	{1, maximum (7)}

[1] Incorrectly reported as 3 dBm by RIOT. The true maximum transmit power of the AT86RF215 is 15.5 dBm [66, p. 191]. See: `https://forum.riot-os.org/t/changing-the-tx-power-of-the-at86rf215`

[2] During the wireless measurements, other radio interference, e.g. by Wi-Fi traffic, was low, since the campus was relatively empty due to the corona virus pandemic and the experiments were conducted in the late evening or night (after 8 p.m.).

Several parameters can be expected to affect the performance of our system, such as other loads on the network or servers or the size of the firmware image being transmitted to the IoT devices. See Table 5.1 for an overview of the relevant parameters. We vary only those parameters that we are interested in evaluating. They are highlighted in Table 5.1. Firstly and most importantly, we vary the transport mechanism used, i.e. CoAP or MQTT-SN. Secondly, we vary the block size used during transmission to find out which effect it has on our evaluation metrics and which block size performs best. Thirdly, we vary the number of devices being updated at the same time to evaluate the scalability of the SUIT implementation. At the time of writing, the maximum possible number of nodes usable for these experiments is seven nodes.[3] Finally, to better isolate the effects of the parameters being varied, we conduct measurements first over Ethernet, which provides a more stable and reproducible network, and then over wireless radio, which is closer to the real environment in which SUIT may be used. All other parameters are fixed throughout our evaluation.

The reasons for the choice of MQTT-SN implementations were already discussed in Section 4.1.1. The CoAP implementations were chosen as suggested in the `README` of the pre-existing SUIT implementation in RIOT.[4] For a full list of all software used including version numbers, see Appendix A.1.

5.2 Methods

In this section, we describe how we conducted our evaluation, e.g. how the data is gathered and which tools are used to analyse it. The raw data, program files, DES-Cript files as well as more detailed descriptions of our methods for the purpose of reproducibility are given at `https://github.com/vera/masters-thesis`.

Memory Usage

To evaluate the memory usage of SUIT in RIOT, we compile our binaries optimized for code size using the flag `-Os`, which is standard for RIOT, inside the Docker container `riot/riotbuild`. The binaries are compiled for the testbed boards (Nucleo F767-ZI) and include the modules of the GNRC network stack necessary for wireless communication over IEEE 802.15.4. We use the C standard library implementation Picolibc[5] instead of the default Newlib[6] because of its reduced memory footprint. Using this setup, the results we achieve for the CoAP version are very similar to those reported by Zandberg et al. in their evaluation [46]. The transmit and receive buffers of `emcute` are left at the default size of 512 B each.

To determine the RAM and flash memory usages of the compiled binaries, we use the binary size profiler Bloaty[7]. Bloaty provides a fine-grained memory usage split down to compile units (i.e. files) or symbols (i.e. function or variable names). It also allows the definition

[3] One of the eight testbed nodes could not be used due to hard faults that occur at the initial firmware booting, presumably due to problems with the connection to the attached EEPROM.

[4] `https://github.com/RIOT-OS/RIOT/blob/80e14e88a10d0795b5d8416edd564bb09f64ae38/examples/suit_update/README.md`

[5] `https://github.com/picolibc/picolibc`

[6] `https://sourceware.org/newlib`

[7] `https://github.com/google/bloaty`

of custom categorization using regular expressions that are applied on the compile unit or symbol names. For example, all file paths containing “emcute” can be sorted into the category “mqtt-sn” to obtain the total memory usage related to the MQTT-SN component.

Network Traffic Volume

The network traffic is captured in a local setup of a Nucleo F767-ZI connected to a laptop computer via Ethernet using Wireshark[8]. The resulting capture file is then analysed using a self-written Python program[9] which uses Pyshark[10], a wrapper for Wireshark's command line equivalent `tshark`, to dissect the packets and then calculates total for each involved protocol and each packet direction (i.e. server/broker to device or device to server/broker). Because the update's network traffic is more or less deterministic when there is no random packet loss, a single capture is sufficient to evaluate the network traffic volume.

Total Duration of the Firmware Update and Energy Consumption

The duration and energy consumption measurements are run in the MIoT Lab, as described above. The MQTT-SN/CoAP server is run on a virtual server that is reachable from the Nucleo boards via the x86 nodes. The connection is established either via Ethernet or via a 6LoWPAN border router[11] running on one of the Nucleo boards. During the experiments, the CPU and RAM usages of the virtual server are logged.

Both duration and power usage are measured at the same time. The measurements are repeated 30 times for each parameter combination. The power usage readings are logged by the ESP32 which is connected to two TI INA3221 current and voltage monitors with three channels each, so they can separately monitor the power usage of the Nucleo board, its transceivers and environmental sensors. The monitors are configured to use a conversion time of 140 µs for both the shunt and bus voltage and to average 64 samples for each reading. New power readings are logged every 860 µs. The readings as well as the start and end timestamps are captured by a Python program running on the x86 node which receives the log outputs of the Nucleo board and the ESP32 over serial connections. To finally calculate the energy consumption, the power readings (in W) are integrated over the time period during which the firmware update was running. For the updates run over Ethernet, we use only the power readings of the Nucleo board itself in this calculation. For the updates run over-the-air, we additionally use the power readings of the IEEE 802.15.4 transceiver. The power readings of the other channels, i.e. the other transceivers and sensors, are discarded since they are not relevant for the firmware update process.

5.3 Results

In this section, we describe and discuss the results of our evaluation.

[8] `https://wireshark.org`
[9] `https://github.com/vera/traffic-analysis`
[10] `https://github.com/KimiNewt/pyshark`
[11] We use the example `gnrc_border_router` provided by RIOT.

Memory Usage

The results of the memory usage analysis are shown in Table 5.2. Note that the actual RAM usages of the two variants could be lower, since we did not optimize the stack sizes but left them at their default values, which may be larger than actually required.

The direct comparison of the memory usages shows that our implementation of the MQTT-SN transport mechanism requires 0.4 KiB more flash memory and 1.2 KiB more RAM than the pre-existing implementation of the CoAP transport mechanism. The slight increase in flash memory usage is due to a slightly larger code size (+0.17 KiB) of the functions required for the Over-The-Air (OTA) updates, e.g. the MQTT-SN PUBLISH handlers (see Section 4.2.2) and the functions required for the usage of thread flags. Thread flags are used in our MQTT-SN transport mechanism code as well as `emcute`'s code to communicate events between threads. The increase in RAM usage is due to RAM used by `emcute`, mainly for the transmit and receive buffers (512 B each by default), which are statically allocated in the BSS section. In contrast, the transmit buffer (256 B by default) and receive buffer (sized according to the requested block size) used by `nanocoap` are allocated on the stack of the transport mechanism's thread.

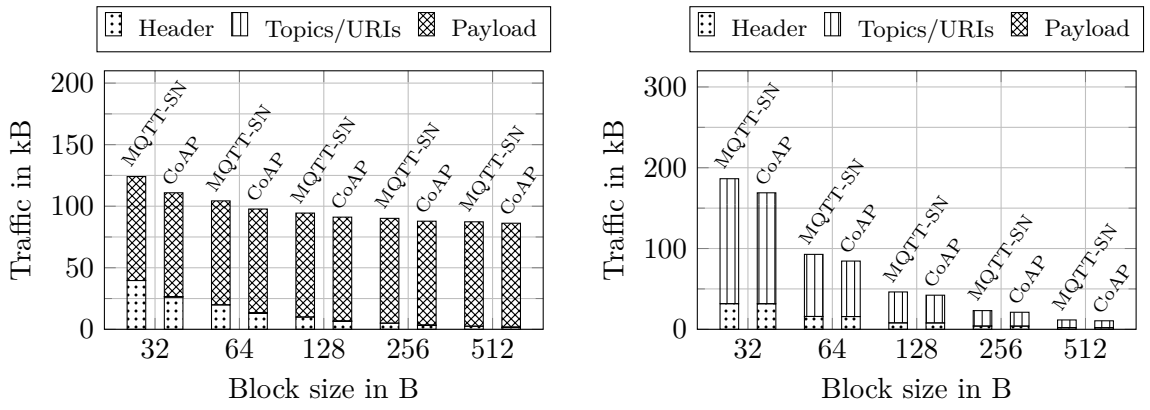
The Nucleo boards that we use in our testbed are less constrained in terms of flash memory and RAM than others, since they are part of STMicroelectronics' "high-performance" product line. This puts them beyond the least constrained class defined by RFC 7228 [12] (Class 2 with ~50 KiB RAM and ~250 KiB flash memory). However, it can be seen that our MQTT-SN variant can also be run on more constrained devices (up to approximately Class 1 with ~10 KiB RAM and ~100 KiB flash memory), same as the pre-existing CoAP variant. Thus, the implementation fulfils the goal set by the SUIT standard to be suitable for Class 1 devices.

Network Traffic Volume

We examine the application layer traffic in Figure 5.1. In both cases, the payload sent from the server or broker to the device is equal to the size of the update manifest and firmware image as listed in Table 5.1 (approximately 85 kB). In addition to the payload

Table 5.2: Flash memory and RAM usage of the SUIT application using the CoAP or the MQTT-SN transport mechanism.

	Flash memory			**RAM**		
Component	CoAP	MQTT-SN	Diff.	CoAP	MQTT-SN	Diff.
Core	28.6 KiB	30.0 KiB	+1.4 KiB	4.6 KiB	4.6 KiB	0
Network	29.3 KiB	28.8 KiB	−0.5 KiB	6.2 KiB	6.2 KiB	0
CoAP/MQTT-SN	2.2 KiB	1.8 KiB	−0.5 KiB	0	1.2 KiB	+1.2 KiB
Crypto	6.2 KiB	6.2 KiB	0	96 B	96 B	0
COSE & CBOR	2.1 KiB	2.0 KiB	−0.1 KiB	0	0	0
SUIT	3.1 KiB	3.0 KiB	−0.0 KiB	48 B	48 B	0
OTA	2.5 KiB	2.6 KiB	+0.2 KiB	8.4 KiB	8.4 KiB	0
Total	74 KiB	74.4 KiB	+0.4 KiB	19.3 KiB	20.5 KiB	+1.2 KiB



a) Broker/server to device.

b) Device to broker/server.

Figure 5.1: Network traffic volume of a full firmware update using SUIT over MQTT-SN or CoAP: Only the relevant network traffic caused by the application layer protocols and payloads is shown.

itself, there is overhead traffic, i.e. protocol headers, MQTT-SN topic names and CoAP URIs. As expected, the amount of overhead traffic decreases approximately linearly with the increased block size.

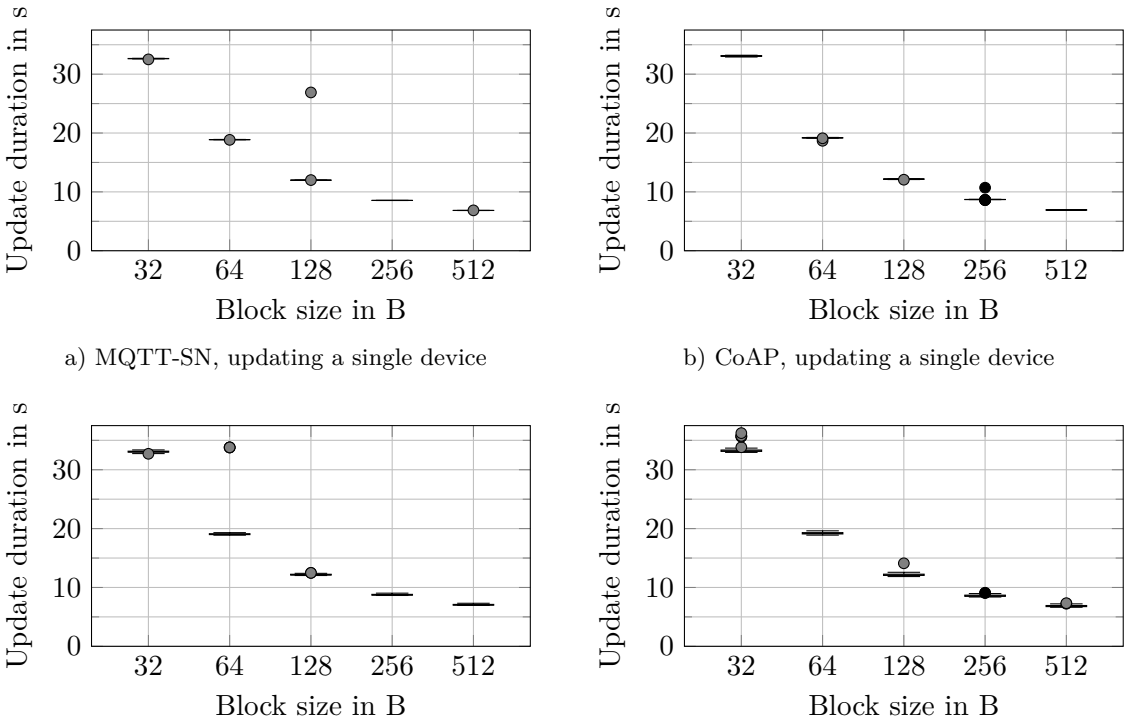
It can be seen that MQTT-SN causes slightly more traffic than CoAP. The larger the block size, the smaller the difference is. For a block size of 32 B, the difference in traffic volume between MQTT-SN and CoAP is the largest: MQTT-SN causes 13.2 kB and 17.4 kB more traffic from broker to device and device to broker, respectively. For a block size of 512 B, MQTT-SN causes only 1.2 kB and 1.1 kB more traffic from broker to device and device to broker, respectively. This is because the sizes of the headers that are sent per block are

Table 5.3: Traffic caused by MQTT-SN and CoAP headers: Note that the CoAP header sizes are specific to the CoAP options used here (e.g. number of URI path components).

	Direction	
Protocol	Client → server	Server → client
MQTT-SN	7 B (SUBSCRIBE) + 5 B (PUBACK) = 12 B	7 B (SUBACK) + 7 B (PUBLISH) = 14 B
CoAP	11 B (GET request)	9 B (GET response)

Table 5.4: Number of packets sent during a full firmware update over MQTT-SN and CoAP.

	Block size				
Protocol	32 B	64 B	128 B	256 B	512 B
MQTT-SN	10569	5293	2653	1333	673
CoAP	5282	2642	1322	662	332



a) MQTT-SN, updating a single device

b) CoAP, updating a single device

c) MQTT-SN, updating seven devices in parallel

d) CoAP, updating seven devices in parallel

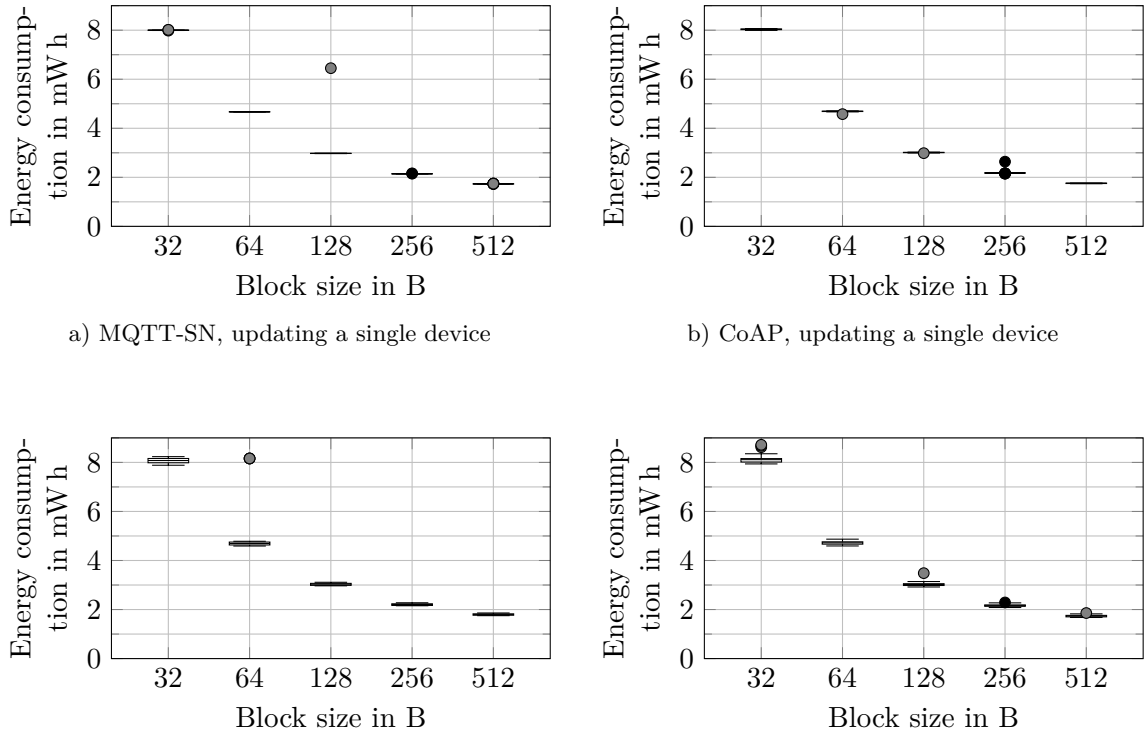
Figure 5.2: Duration of a full firmware update over Ethernet using MQTT-SN or CoAP.

smaller for CoAP than for MQTT-SN (see Table 5.3). As expected, MQTT-SN also sends twice as many packets per block as CoAP does (see Table 5.4). Thus, the amount of traffic caused by lower layer protocol headers such as IEEE 802.15.4 and 6LoWPAN, which we do not examine here, is doubled.

Total Duration of the Firmware Update and Energy Consumption

The measured total durations of the firmware updates over Ethernet are shown as boxplots in Figure 5.2. It can be seen that there is little variation between the 30 repetitions of each parameter combination. Firstly, Figure 5.2a and Figure 5.2b show the results for the firmware update of a single device. The standard deviations are below 1 %, except for the combinations (Ethernet, MQTT-SN, 128 B, 1 device) and (Ethernet, CoAP, 256 B, 1 device), where there are outliers with significantly increased durations. Secondly, Figure 5.2c and Figure 5.2d show the results for the firmware update of seven devices in parallel. Here, the standard deviations of the 30 repetitions are slightly larger (up to 4.4 %), except for the combination (Ethernet, MQTT-SN, 64 B, 7 devices), where two outliers were measured. The reason for these outliers could not be determined definitively from the experiment logs. As the outliers lie approximately 15 s above the other measurements, which is the MQTT-SN's default timeout duration, it can be assumed that they are due to a single lost MQTT-SN message.

Comparing the different measurements, it can be seen that, firstly, as expected, the update



a) MQTT-SN, updating a single device

b) CoAP, updating a single device

c) MQTT-SN, updating seven devices in parallel

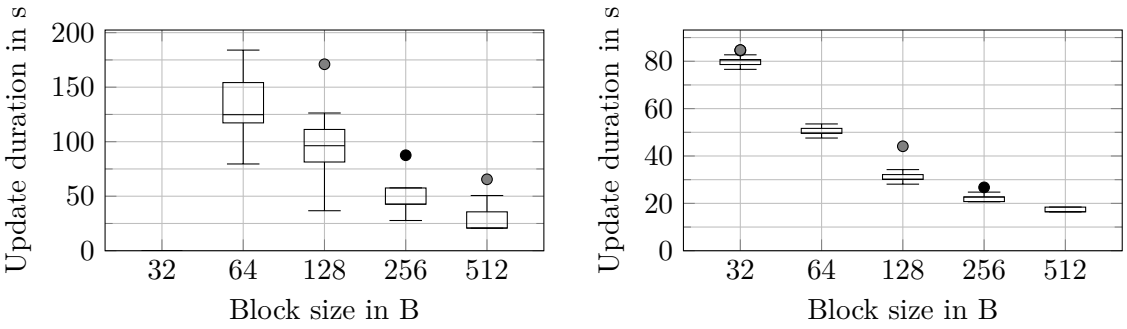
d) CoAP, updating seven devices in parallel

Figure 5.3: Energy consumption of the IoT board during a full firmware update over MQTT-SN and CoAP.

duration decreases with the block size. The decrease is not fully linear because the larger block sizes do not speed up all steps of the firmware update; e.g. the time required for the validation of manifest and signatures remains the same. Secondly, even though a higher total duration could be expected for MQTT-SN due to the higher traffic volume (see Section 5.3), there is no significant difference between the average update durations measured for MQTT-SN and CoAP. Thirdly, when transmitting the updates over Ethernet, there is also no significant difference between the average update durations measured for a single device and seven devices in parallel.

During the parallel update of seven devices, the total RAM usage on the virtual machine running the MQTT-SN/CoAP server remains at approximately 9 %. In the CPU usage, there are peaks that coincide with the timings of the firmware updates. These peaks go up to 92.4 %, but do not last more than 5 seconds, i.e. not throughout the entire update transmission. Except during these peaks and during the waiting times between updates, the CPU usage is at 5 % to 15 %. Thus, the server does not appear to act as a performance bottleneck during the parallel update experiments.

The measured energy consumption during the firmware updates run over Ethernet are shown as boxplots in Figure 5.3. As expected, it can be seen that there is a strong correlation between the measured energy consumption and the measured update duration. Thus, the



a) MQTT-SN, updating a single device

b) CoAP, updating a single device

Figure 5.4: Duration of a full firmware update over IEEE 802.15.4 using MQTT-SN or CoAP.

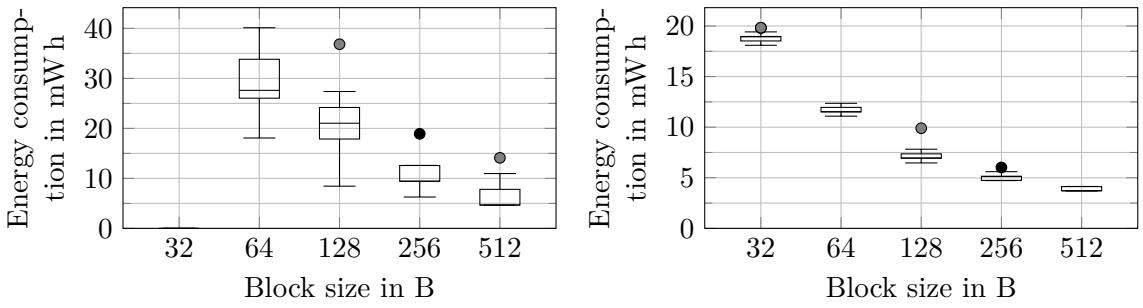
same observations can be made here as for Figure 5.2: The energy consumption decreases slightly less than linearly with the block size. There is no significant difference between the average energy consumption measured for MQTT-SN and CoAP.

Finally, the duration measurements for the wireless update of a single device (using IEEE 802.15.4/ 6LoWPAN) can be seen in Figure 5.4. In these experiments, testbed node 7 is used as the border router. Testbed node 10 ("TestNode") is used as the SUIT node. Node 10 is positioned in an adjoining room to node 7, i.e. the two nodes are relatively close to each other. Using the 2.4 GHz band, the average RSSI between the two nodes is measured to be 77.27 dBm.

A complication with the wireless MQTT-SN measurements was caused by an unclear point in the protocol specification: It states that only messages unicasted to the MQTT-SN gateway by a client are retransmitted if no acknowledgment is received before a timeout [26, p. 25]. There is no retransmission procedure specified for messages unicasted by the gateway to a client, such as the firmware blocks in our case. This is likely because the MQTT-SN gateway is meant to be used in conjunction with an MQTT broker, which would handle all retransmissions of unacknowleged messages. Thus, the gateway does not need to retransmit lost messages itself. However, the RSMB server implementation of MQTT-SN does not implement a standalone gateway that connects to a separate broker. The RSMB integrates an MQTT and MQTT-SN broker in the same application. When a client connects to the

Table 5.5: Average update durations of both wired and wireless updates over MQTT-SN and CoAP.

	MQTT-SN			**CoAP**		
Block size	Wired	Wireless	Diff.	Wired	Wireless	Diff.
32 B	32.65 s	n/a	n/a	33.09 s	80.45 s	+143.13 %
64 B	18.88 s	131.6 s	+597.03 %	19.17 s	50.35 s	+162.65 %
128 B	12.48 s	92.88 s	+644.23 %	12.17 s	31.36 s	+157.68 %
256 B	8.55 s	46.43 s	+443.04 %	8.76 s	22.18 s	+153.2 %
512 B	6.84 s	30.15 s	+340.79 %	6.94 s	17.26 s	+148.7 %



a) MQTT-SN, updating a single device

b) CoAP, updating a single device

Figure 5.5: Energy consumption of the IoT board (including transceiver) during a full firmware update over IEEE 802.15.4 using MQTT-SN or CoAP.

RSMB using MQTT-SN, there is no connection made internally to the MQTT broker. Thus, the broker actually does not retransmit any lost messages, since it is not involved in the transaction. This means that when a block is lost in transmission, the update fails. Additionally, there appears to be a bug in RIOT affecting IEEE 802.15.4 retransmissions over the AT86RF215 transceiver: A sniffer trace also showed no such retransmissions in case of a missing acknowledgement. Due to time constraints, this could not be investigated further.

As expected due to the lower throughput, the wireless updates are slower than the wired updates. We compare the average update durations side-by-side in Table 5.5. Over CoAP, the wireless updates are slower by 153 % on average when compared to the wired updates. Over MQTT-SN, the wireless updates are slower by 506 % on average. Thus, although there was no significant difference between CoAP and MQTT-SN when running wired updates, MQTT-SN performs much worse than CoAP when updating over the air. The main reason for this is the much longer default timeout duration for retransmissions of 15 s [26, p. 27]. This means that every time an MQTT-SN message is lost, the update duration increases by at least 15 s. Thus, the performance is worst for the smallest block sizes where the most MQTT-SN messages are transmitted, each of which can potentially get lost and cause a timeout. Additionally, the chance of an update-stopping timeout occurring (after three

Table 5.6: Effect of block size increases on average update durations of both wired and wireless updates over MQTT-SN and CoAP.

	MQTT-SN		**CoAP**	
Block sizes	Wired	Wireless	Wired	Wireless
32 B → 64 B	−42.18 %	n/a	−42.07 %	−37.42 %
64 B → 128 B	−33.9 %	−29.42 %	−36.52 %	−37.72 %
128 B → 256 B	−31.49 %	−50.01 %	−28.02 %	−29.27 %
256 B → 512 B	−20 %	−35.06 %	−20.78 %	−22.18 %

unsuccessful retransmissions) is higher the smaller the block size is. Due to this, none of 60 repetitions of a wireless update over MQTT-SN with a block size of 32 B could be completed successfully. In contrast, CoAP uses exponentially increasing timeout durations with a default starting value of 2 s to 3 s and thus retransmits much faster.

The effect of increasing the block size is very similar to the one observed for wired updates: A doubling of the block size leads to a decrease of the average update duration of around 33 % on average (see Table 5.6). As discussed previously, the decrease is less than 50 % because the increased block size speeds up the transmission of the update, but not other steps, e.g. the validation of manifest and signatures, and writing the update image to flash memory. It must be noted that the results also show that the speedup effect decreases with each block size doubling. Considering wireless transmission, it can be expected that the speedup effect is smaller because starting from a block size of 64 B, the blocks must be fragmented by 6LoWPAN due to IEEE 802.15.4's maximum frame size of 127 B. This fragmentation adds additional overhead in terms of computation and traffic. Over CoAP, it can be seen that the speedup effect of the jump from 32 B to 64 B is indeed smaller than it is for wired transmission, while the other jumps show a similar speedup as for wired transmission. Over MQTT-SN, the block size increases show a larger effect for wireless than for wired transmission. This may again be related to the long retransmission timeouts; if there are fewer MQTT-SN messages to transmit, a smaller number of retransmission timeouts is likely to occur. Thus, the speedup effect is boosted.

As expected, the energy consumption of a wireless update is also higher than the energy consumption of an update over Ethernet (see Figure 5.5). Again, there is a strong correlation between the update duration and energy consumption.

For wireless updates of multiple devices at once, we could conduct only preliminary experiments which showed that the GNRC border router performs badly under load: We observed update success rates of less than 50 %. In a minimal test setup consisting of one border router and two wireless SUIT nodes, updating the nodes one after another was actually significantly faster than updating them in parallel. The border router logs showed frequent acknowledgement timeouts as well as sending delays due to a busy channel, which arise from the wireless channel being shared between all updated devices. It may be possible to improve the performance of the border router under heavy load by using the `gnrc_netif_pktq` module, which implements a queue for outgoing packets that could not be transmitted successfully on the first try. Additionally, the reliability of parallel over-the-air updates could be increased by using a Media Access Control (MAC) protocol based on Time Division Multiple Access (TDMA), and the shared channel could be better utilized using multicast to efficiently transmit the update to multiple devices at once.

5.4 Conclusion

In conclusion, the evaluation of our implementation shows that it fulfils the goal that was previously set, i.e. achieving the same or better performance in comparison with the CoAP transport mechanism, with the exception of the issues discovered when transmitting updates over the air.

The evaluation of the memory usage of our implementation shows that it requires only slightly more flash memory (+0.4 KiB) and RAM (+1.2 KiB) than the CoAP transport

mechanism, mainly due to the code size and the transmit and receive buffers of the `emcute` library. Thus, it is suitable for constrained Class 1 devices.

With regard to network traffic on the application layer and up, our MQTT-SN transport mechanism causes slightly more traffic because the MQTT-SN headers sent per block are larger than the CoAP headers by 6 B. Thus, the difference in traffic volume is almost negligible for larger block sizes. In our design, where one MQTT-SN topic is used per block, MQTT-SN sends twice as many packets as CoAP, and thus the lower-layer header traffic is also doubled.

However, the evaluation of the update duration when transmitting over Ethernet shows that our implementation is not slower than the CoAP transport mechanism. Both when updating a single device and when updating seven devices in parallel, there is no significant difference between the measured update durations. There is also no significant difference between the update durations for updating a single device and updating seven devices in parallel; thus, both transport mechanisms scale very well for up to seven devices. Increased block sizes lead to shorter update durations; when doubling the block size, the speedup is around 33 % on average. However, the speedup effect decreases with each block size doubling, i.e. the update duration begins to plateau after a few doublings. The speedup is less than 50 % because some parts of the update are not sped up by the increased block size, e.g. signature verification. Additionally, the time required to write the update image to flash memory is a lower bound on the update duration. For wireless transmission, the added overhead of fragmentation decreases the speedup effect. As expected, our evaluation shows that the wireless transmission is slower than the wired transmission. Over CoAP, it is slower by 153 % on average. Over MQTT-SN, some issues were discovered; it is slower by 506 % on average. The main issue is missing retransmissions due to an unclear specification and the high default timeout duration of 15 s. In all cases, the energy consumption is strongly correlated to the update duration.

Since the evaluation results show no significant difference between CoAP and MQTT-SN with regard to the duration and energy consumption of a firmware update when the updates are run over a wired connection, the MQTT-SN transport mechanism is not inherently less performant than the CoAP transport mechanism. However, it does not handle lost messages as well as CoAP does, and thus performs worse for wireless transmission. Further work on this issue could improve the performance of the MQTT-SN transport mechanism over the air.

The main advantage of MQTT-SN over CoAP, which was discussed in Section 2.4 and Section 3.1 where we compared the protocols generally, remains: It offers better support for sleeping devices because it allows the devices to be in a low-power sleep mode most of the time, during which the broker buffers messages, while CoAP requires the devices to be constantly awake (except for radio duty cycling on the MAC layer) if they want to receive requests. Thus, MQTT-SN allows a lower power consumption than CoAP during the idle times: According to the data sheets, the Nucleo board's STM32 microcontroller draws at most 16.5 µW [67, p. 125] and the ATF86 transceiver typically draws 0.09 µW [66, p. 203] in their lowest power modes. Thus, the low-power modes allow a vastly improved battery life of the IoT device.

CHAPTER 6

Conclusion

In this chapter, we summarize our results and provide pointers for future works.

6.1 Summary

In this work, we investigated the issue of firmware updates for the IoT. Firstly, we summarized the key features of the some of the most used application layer protocols for the IoT: CoAP, MQTT and MQTT-SN. We compared the three protocols. The main difference between CoAP and the MQTT protocols is the communication model used: CoAP uses a request/response model similar to HTTP, while MQTT and MQTT-SN use a publish/subscribe model. The main advantage of the publish/subscribe model is that all communication occurs via a central broker. This allows MQTT-SN to easily support sleeping devices by buffering messages at the broker, which is essential for conserving the limited battery power of IoT devices. However, the transfer of large files such as firmware updates fits CoAP's request/response model more naturally, especially using block-wise transfer.

Secondly, we conducted a literature survey of comparative evaluations of the three application layer protocols with regard to their achieved throughputs, energy consumption, traffic overhead and ability to deal with packet loss. These evaluations show that TCP – which is used by MQTT – is not especially suitable for the IoT due to the connection setup and teardown overhead and its larger header size, even though its sliding window protocol for retransmissions allows it to deal with packet loss better than the UDP-based protocols. Only few previous evaluation results are available for MQTT-SN, but they suggest that it can perform as well as CoAP.

Thirdly, we defined a list of requirements for software update mechanisms in the context of the IoT. We surveyed other software update mechanisms besides SUIT, such as TUF, Uptane, and UpKit, and analysed their main similarities and differences to SUIT. A common feature of all surveyed update mechanisms is the usage of cryptographic signatures for authentication and integrity checking of the update images and metadata. There are differences in the metadata formats and how exactly the cryptographic signatures are used. For example, UpKit always uses two cryptographic signatures; one by the firmware vendor and one by the update server that binds the update to a random nonce generated by the IoT device to prevent replay attacks. In contrast, SUIT uses only a single signature and

uses optional expiration timestamps to prevent the replaying of an outdated update. Overall, SUIT offers the most general framework for software updates through the command sequences that are included in the update manifests and can flexibly specify the steps of the fetching, validation and installation process according to the use case.

Finally, we described the design and implementation of a new transport mechanism using MQTT-SN for firmware updates over SUIT in RIOT. Previously, only a transport mechanism using CoAP was available. We evaluated the implementation firstly with regard to its flash memory and RAM requirements and network traffic volume. Our evaluation showed that our new transport mechanism uses a similar amount of flash memory and RAM as the CoAP transport mechanism. Because of the design choice of using a separate MQTT-SN topic for each firmware block, our new transport mechanism causes slightly more network traffic due to application layer header overhead. Secondly, we evaluated the implementation in the MIoT Lab testbed at the Otto von Guericke University Magdeburg. At the time of writing, the testbed consisted of a maximum of seven usable nodes, which we used for scalability tests. The testbed experiments showed that over Ethernet, there is no significant difference of update duration and energy consumption between the two transport mechanisms, i.e. MQTT-SN performs as well as CoAP. Thus, the implementation fulfils its main goals. However, during over-the-air experiments using IEEE 802.15.4/6LoWPAN, we discovered some issues with the MQTT-SN protocol specification that caused a relatively poor performance of MQTT-SN: The specification is unclear on the issue of who is responsible for retransmitting lost messages, and defines a relatively long default retransmission timeout of 10 s to 15 s. Thus, further work is necessary to bring MQTT-SN's performance over wireless connections up to CoAP's level.

6.2 Future Work

To improve upon this work, the performance issues of MQTT-SN when wirelessly transmitting updates as described in Section 5.3 must be further investigated. Additionally, the initial scalability experiments for over-the-air updates showed performance issues, which should be investigated and fixed if possible. In particular, the usage of multicast to deliver the update to multiple nodes at once could greatly improve the performance.

To achieve better performance, some design choices of the MQTT-SN transport mechanism could be reconsidered. For example, the traffic of REGISTER messages could be reduced. In the current design, one REGISTER is sent for each firmware manifest and image block transmitted because each block uses a different topic. Instead, a single topic could be used. This would require the update image and manifest to be published "live" during the update, since retained messages cannot be used. This places additional load on the host which publishes the update files. However, this may be a viable option if it were possible to update a large amount of nodes at once. Alternatively, a MQTT-SN broker implementation could be extended with a feature similar to CoAP's block-wise transfer.

The scalability measurements presented were limited by the number of available nodes in the testbed, which is seven at the time of writing. Once more testbed nodes are rolled out, more extensive scalability experiments could be conducted.

Finally, future works could build on this work by implementing and evaluating further transport mechanisms using other application layer protocols for the IoT.

Bibliography

[1] Ronald Eikenberg. IP-Kameras von Aldi als Sicherheits-GAU. Heise Security, January 2016. https://www.heise.de/security/meldung/IP-Kameras-von-Aldi-als-Sicherheits-GAU-3069735.html (last visited on Apr 01, 2021).

[2] Moshe Kol and Shlomi Oberman. CVE-2020-11896 RCE and CVE-2020-11898 Info Leak (Ripple20). Technical report, JSOF Research Lab, June 2020. https://www.jsof-tech.com/jsof_ripple20_technical_whitepaper_june20/.

[3] Ang Cui, Michael Costello, and Salvatore Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. 20th Annual Network & Distributed System Security Symposium, February 2013. https://www.ndss-symposium.org/ndss2013/ndss-2013-programme/when-firmware-modifications-attack-case-study-embedded-exploitation.

[4] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[5] Bundesamt für Sicherheit in der Informationstechnik, Bonn, Germany. *IT-Grundschutz-Kompendium. SYS.4.4: Allgemeines IoT-Gerät.* Reguvis Fachmedien GmbH, February 2020. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium_Einzel_PDFs/07_SYS_IT_Systeme/SYS_4_4_Allgemeines_IoT_Geraet_Edition_2020.pdf.

[6] Michael Fagan, Katerina N. Megas, Karen Scarfone, and Matthew Smith. *IoT Device Cybersecurity Capability Core Baseline.* National Institute of Standards and Technology, Gaithersburg, MD, USA, May 2020. https://doi.org/10.6028/NIST.IR.8259A.

[7] Directive (EU) 2019/771 of the European Parliament and of the Council of 20 May 2019 on certain aspects concerning contracts for the sale of goods, amending Regulation (EU) 2017/2394 and Directive 2009/22/EC, and repealing Directive 1999/44/EC. *Official Journal of the European Union*, 62, May 2019. https://eur-lex.europa.eu/eli/dir/2019/771/oj.

[8] Bundesministerium der Justiz und für Verbraucherschutz. Entwurf eines Gesetzes zur Regelung des Verkaufs von Sachen mit digitalen Elementen und anderer Aspekte des Kaufvertrags, December 2020. https://bmjv.de/SharedDocs/Gesetzgebungsverfahren/DE/Warenkaufrichtlinie.html.

[9] Bruce Schneier. The Internet of Things Is Wildly Insecure – And Often Unpatchable, January 2014. https://www.schneier.com/essays/archives/2014/01/the_internet_of_thin.html (last visited on Oct 17, 2020).

[10] Iulia Ion, Rob Reeder, and Sunny Consolvo. "...no one can hack my mind": Comparing Expert and Non-Expert Security Practices. In *Eleventh Symposium On Usable Privacy and Security ({SOUPS} 2015)*, pages 327–346, 2015.

[11] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 195–212. Institute of Electrical and Electronics Engineers (IEEE), 2017. https://doi.org/10.1109/SP.2017.14.

[12] Carsten Bormann, Mehmet Ersue, and Ari Keränen. Terminology for Constrained-Node Networks. RFC 7228, May 2014. https://rfc-editor.org/rfc/rfc7228.txt.

[13] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. A Firmware Update Architecture for Internet of Things. RFC 9019, April 2021. https://rfc-editor.org/rfc/rfc9019.txt.

[14] Brendan Moran, Hannes Tschofenig, Henk Birkholz, and Koen Zandberg. A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest. Internet-Draft draft-ietf-suit-manifest-12, Internet Engineering Task Force, February 2021. Work in Progress. https://datatracker.ietf.org/doc/html/draft-ietf-suit-manifest-12.

[15] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. A Manifest Information Model for Firmware Updates in IoT Devices. Internet-Draft draft-ietf-suit-information-model-11, Internet Engineering Task Force, April 2021. Work in Progress. https://datatracker.ietf.org/doc/html/draft-ietf-suit-information-model-11.

[16] Emmanuel Baccelli, Oliver Hahm, Matthias Wählisch, Mesut Gunes, and Thomas Schmidt. RIOT: One OS to Rule Them All in the IoT. [Research Report] RR-8176, INRIA, 2012. https://hal.inria.fr/hal-00768685v3.

[17] Maria Rita Palattella, Nicola Accettura, Xavier Vilajosana, Thomas Watteyne, Luigi Alfredo Grieco, Gennaro Boggia, and Mischa Dohler. Standardized Protocol Stack for the Internet of (Important) Things. *IEEE Communications Surveys & Tutorials*, 15(3):1389–1406, 2012. https://doi.org/10.1109/SURV.2012.111412.00158.

[18] Zhengguo Sheng, Shusen Yang, Yifan Yu, Athanasios V. Vasilakos, Julie A. Mccann, and Kin K. Leung. A Survey on the IETF Protocol Suite for the Internet of Things: Standards, Challenges, and Opportunities. *IEEE Wireless Communications*, 20(6):91–98, 2013. https://doi.org/10.1109/MWC.2013.6704479.

[19] Gabriel Montenegro, Jonathan Hui, David Culler, and Nandakishore Kushalnagar. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, September 2007. https://rfc-editor.org/rfc/rfc4944.txt.

[20] Pascal Thubert and Jonathan Hui. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282, September 2011. https://rfc-editor.org/rfc/rfc6282.txt.

[21] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Pro-

tocol (CoAP). RFC 7252, June 2014. https://rfc-editor.org/rfc/rfc7252.txt.

[22] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015. https://rfc-editor.org/rfc/rfc7540.txt.

[23] Ramon Caceres and Liviu Iftode. The Effects of Mobility on Reliable Transport Protocols. In *14th International Conference on Distributed Computing Systems*, pages 12–20. IEEE, 1994. https://doi.org/10.1109/ICDCS.1994.302385.

[24] Carsten Bormann and Zach Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, August 2016. https://rfc-editor.org/rfc/rfc7959.txt.

[25] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. *MQTT Version 5.0.* OASIS Standard, March 2019. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.

[26] Andy Stanford-Clark and Hong Linh Truong. *MQTT For Sensor Networks (MQTT-SN) Protocol Specification (Version 1.2).* International Business Machines Corporation (IBM), November 2013. https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf.

[27] Michael Koster, Ari Keränen, and Jaime Jimenez. Publish-Subscribe Broker for the Constrained Application Protocol (CoAP). Internet-Draft draft-ietf-core-coap-pubsub-09, Internet Engineering Task Force, September 2019. Work in Progress. https://datatracker.ietf.org/doc/html/draft-ietf-core-coap-pubsub-09.

[28] Eclipse IoT Working Group, AGILE IoT, Institute of Electrical and Electronics Engineers, and Open Mobile Alliance. IoT Developer Survey Results, April 2018. https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2018.pdf (last visited on Feb 19, 2021).

[29] Jasenka Dizdarević, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration. *ACM Computing Surveys (CSUR)*, 51(6):1–29, 2019. https://doi.org/10.1145/3292674.

[30] Markel Iglesias-Urkia, Adrián Orive, and Aitor Urbieta. Analysis of CoAP Implementations for Industrial Internet of Things: A Survey. *Procedia Computer Science*, 109:188–195, 2017. https://doi.org/10.1016/j.procs.2017.05.323.

[31] Pegah Nikbakht Bideh, Jonathan Sönnerup, and Martin Hell. Energy Consumption for Securing Lightweight IoT Protocols. In *Proceedings of the 10th International Conference on the Internet of Things*, pages 1–8, 2020. https://doi.org/10.1145/3410992.3411008.

[32] Matteo Collina, Marco Bartolucci, Alessandro Vanelli-Coralli, and Giovanni Emanuale Corazza. Internet of Things Application Layer Protocol Analysis over Error and Delay Prone Links. In *2014 7th Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pages 398–404. IEEE, 2014. https://doi.org/10.1109/ASMS-SPSC.2014.6934573.

[33] Dae-Hyeok Mun, Minh Le Dinh, and Young-Woo Kwon. An Assessment of Internet of

Things Protocols for Resource-Constrained Applications. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 555–560. IEEE, 2016. https://doi.org/10.1109/COMPSAC.2016.51.

[34] Stefan Mijovic, Erion Shehu, and Chiara Buratti. Comparing Application Layer Protocols for the Internet of Things via Experimentation. In *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 1–5. IEEE, 2016. https://doi.org/10.1109/RTSI.2016.7740559.

[35] Anna Larmo, Felipe Del Carpio, Pontus Arvidson, and Roman Chirikov. Comparison of CoAP and MQTT Performance Over Capillary Radios. In *2018 Global Internet of Things Summit (GIoTS)*, pages 1–6, 2018. https://doi.org/10.1109/GIOTS.2018.8534576.

[36] Cenk Gündoğan, Peter Kietzmann, Martine Lenders, Hauke Petersen, Thomas C Schmidt, and Matthias Wählisch. NDN, CoAP, and MQTT: A Comparative Measurement Study in the IoT. In *Proceedings of the 5th ACM Conference on Information-Centric Networking*, pages 159–171, 2018. https://doi.org/10.1145/3267955.3267967.

[37] Cédric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frédéric Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. Milan, Italy, December 2015. https://hal.inria.fr/hal-01213938.

[38] Yuang Chen and Thomas Kunz. Performance Evaluation of IoT Protocols under a Constrained Wireless Access Network. In *2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*, pages 1–7. IEEE, 2016. https://doi.org/10.1109/MoWNet.2016.7496622.

[39] Francisco Javier Acosta Padilla, Emmanuel Baccelli, Thomas Eichinger, and Kaspar Schleiser. The Future of IoT Software Must be Updated. *IAB Workshop on Internet of Things Software Update (IoTSU)*, June 2016.

[40] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251. Association for Computing Machinery, 2017. https://doi.org/10.1145/3132747.3132786.

[41] Jan Bauwens, Peter Ruckebusch, Spilios Giannoulis, Ingrid Moerman, and Eli De Poorter. Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles. *IEEE Communications Magazine*, 58(2):35–41, 2020. https://doi.org/10.1109/MCOM.001.1900125.

[42] Luca Mottola, Gian Pietro Picco, and Adil Amjad Sheikh. FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks. In *European Conference on Wireless Sensor Networks*, pages 286–304. Springer, 2008.

[43] International Organization for Standardization. ISO/IEC 27000:2018 – Information technology – Security techniques – Information security management systems

– Overview and vocabulary, February 2018. https://standards.iso.org/ittf/PubliclyAvailableStandards/c073906_ISO_IEC_27000_2018_E.zip.

[44] Dustin Frisch, Sven Reißmann, and Christian Pape. An Over the Air Update Mechanism for ESP8266 Microcontrollers. *ICSNC 2017: The Twelfth International Conference on Systems and Networks Communications*, 2017.

[45] Hannes Tschofenig and Stephen Farrell. Report from the Internet of Things Software Update (IoTSU) Workshop 2016. RFC 8240, September 2017. https://rfc-editor.org/rfc/rfc8240.txt.

[46] Koen Zandberg, Kaspar Schleiser, Francisco Acosta, Hannes Tschofenig, and Emmanuel Baccelli. Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check. *IEEE Access*, 7:71907–71920, 2019. https://doi.org/10.1109/ACCESS.2019.2919760.

[47] Klint Finley. Nest's Hub Shutdown Proves You're Crazy to Buy Into the Internet of Things, May 2016. https://www.wired.com/2016/04/nests-hub-shutdown-proves-youre-crazy-buy-internet-things/ (last visited on Mar 03, 2021).

[48] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable Key Compromise in Software Update Systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 61–72, 2010. https://doi.org/10.1145/1866307.1866315.

[49] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. A Look in the Mirror: Attacks on Package Managers. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 565–574, 2008. https://doi.org/10.1145/1455770.1455841.

[50] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure Software Updates: Disappointments and New Challenges. In *HotSec '06: 1st USENIX Workshop on Hot Topics in Security*, 2006.

[51] Trishank Karthik, Akan Brown, Sebastien Awwad, Damon McCoy, Russ Bielawski, Cameron Mott, Sam Lauzon, André Weimerskirch, and Justin Cappos. Uptane: Securing Software Updates for Automobiles. In *International Conference on Embedded Security in Car*, pages 1–11, 2016.

[52] Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer. UpKit: An Open-Source, Portable, and Lightweight Update Framework for Constrained IoT Devices. *39th IEEE International Conference on Distributed Computing Systems (ICDCS)*, July 2019.

[53] Wassim Itani, Ayman Kayssi, and Ali Chehab. PETRA: A Secure and Energy-Efficient Software Update Protocol for Severely-Constrained Network Devices. In *Q2SWinet '09: Proceedings of the 5th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, pages 37–43, October 2009. https://doi.org/10.1145/1641944.1641952.

[54] Cory Doctorow. Philips Pushes Lightbulb Firmware Update That Locks Out Third-Party Bulbs, December 2015. https://boingboing.net/2015/12/14/philips-pushes-lightbulb-firmw.html (last visited on Mar 10, 2021).

[55] Cory Doctorow. HP Detonates Its Timebomb: Printers Stop Accepting Third Party Ink

En Masse, September 2016. https://boingboing.net/2016/09/19/hp-detonates-its-timebomb-pri.html (last visited on Mar 10, 2021).

[56] Ashley Carman. Smart Lock Vendor Accidentally Bricks Its Own Locks Through Firmware Update, August 2017. https://www.theverge.com/circuitbreaker/2017/8/15/16151798/lockstate-6i-software-update-break-lock (last visited on Mar 10, 2021).

[57] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 5(6):4428–4440, 2018. https://doi.org/10.1109/jiot.2018.2815038.

[58] Brendan Moran, Hannes Tschofenig, Henk Birkholz, and Koen Zandberg. A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest. Internet-Draft draft-ietf-suit-manifest-09, Internet Engineering Task Force, July 2020. Work in Progress. https://datatracker.ietf.org/doc/html/draft-ietf-suit-manifest-09.

[59] Carsten Bormann and Paul E. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049, October 2013. https://rfc-editor.org/rfc/rfc7049.txt.

[60] Jim Schaad. CBOR Object Signing and Encryption (COSE). RFC 8152, July 2017. https://rfc-editor.org/rfc/rfc8152.txt.

[61] The Update Framework Specification (Version: 1.0.17), December 2020. https://github.com/theupdateframework/specification/blob/9d21a28ff143d323014c92c32f235383f3d3f5b6/tuf-spec.md.

[62] N. Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11), November 2018.

[63] Donald E. Eastlake 3rd, Steve Crocker, and Jeffrey I. Schiller. Randomness Requirements for Security. RFC 4086, June 2005. https://rfc-editor.org/rfc/rfc4086.txt.

[64] Silvie Schmidt, Mathias Tausig, Manuel Koschuch, Matthias Hudler, Georg Simhandl, Patrick Puddu, and Zoran Stojkovic. How Little is Enough? Implementation and Evaluation of a Lightweight Secure Firmware Update Process for the Internet of Things. *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security (IoTBDS 2018)*, pages 63–72, 2019.

[65] Kai Kientopf, Marian Buschsieweke, and Mesut Güneş. Technical Report: Designing a Testbed for Wireless Communication Research on Embedded Devices. In *18. GI/ITG KuVS FachGespräch SensorNetze (FGSN 2019)*, pages 41–44, 2019.

[66] Atmel Corporation. Atmel AT86RF215 Datasheet (Revision 42415E), May 2016. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42415-WIRELESS-AT86RF215_Datasheet.pdf (last visited on Jun 01, 2021).

[67] STMicroelectronics. STM32F765xx STM32F767xx STM32F768Ax STM32F769xx

Datasheet (Revision 7), February 2021. https://www.st.com/resource/en/datasheet/stm32f767zi.pdf (last visited on Jun 01, 2021).

Appendix

A.1 List of Software Versions Used

Name and Description	**Version**
Main software	
`aiocoap` (Python CoAP implementation)	0.4.1
`libcoap` (C CoAP implementation)	4.2.1
RIOT (OS for IoT devices)	`https://github.com/RIOT-OS/RIOT` at commit `837b55f` (Feb 1, 2021)
RSMB (MQTT/MQTT-SN broker)	`https://github.com/eclipse/mosquitto.rsmb` at commit `36fd4ba` (Dec 21, 2020) + cherry-picked commit `6fed013` from `https://github.com/eclipse/mosquitto.rsmb/pull/32`
Development	
Docker (Virtualisation tool)	19.03.8
`riot/riotbuild` (Docker image for building RIOT applications)	`sha256-0673aacb9dfd9b0a00f7695b60dbbc0ce48437e9fc354904fde92611dea9fde2`
Evaluation	
Bloaty (Size profiler for binary files)	`https://github.com/google/bloaty` at commit `c41086c` (Nov 19, 2020)
PyShark (Python wrapper for Wireshark)	0.4.3
Python	3.8.5 (locally + testbed server), 3.7.3 (testbed nodes)
Wireshark (Network traffic capture and analysis tool)	3.4.2-1 ubuntu20.04.0+wiresharkdevstable1

A.2 List of Reported Issues and Pull Requests Related to This Thesis

A.2.1 Reported Issues

Description	**Link**	**Reported on**	**Status**
Paho MQTT-SN Gateway			
Gateway assigns non-zero topic ID for wildcard topics	`https://github.com/eclipse/paho.mqtt-sn.embedded-c/issues/221`	Dec 9, 2020	Fixed on May 13, 2021
Crash (segmentation fault) on PUBLISH when `MAX_TOPIC_PAR_CLIENT` is exceeded	`https://github.com/eclipse/paho.mqtt-sn.embedded-c/issues/225`	Jan 18, 2021	Fixed on May 13, 2021
Crash (segmentation fault) when `getaddrinfo` fails	`https://github.com/eclipse/paho.mqtt-sn.embedded-c/issues/229`	Feb 4, 2021	Fixed on May 12, 2021
Gateway sends PUBLISH using topic ID for which the REGISTER was rejected	`https://github.com/eclipse/paho.mqtt-sn.embedded-c/issues/230`	Feb 4, 2021	Fixed on May 13, 2021

A.2.2 Pull Requests

Description	**Link**	**Submitted on**	**Status**
RIOT			
net/emcute: Allow RETAIN flag to be set on incoming PUBLISHs	`https://github.com/RIOT-OS/RIOT/pull/16326`	Apr 13, 2021	Merged on Jun 28, 2021
Paho MQTT-SN Gateway			
Send DISCONNECT when incoming message can't be mapped to a client	`https://github.com/eclipse/paho.mqtt-sn.embedded-c/pull/222`	Dec 11, 2020	Closed on May 27, 2021
mqtt-sn-tools			
Bugfix: Use msg ID 0 for QoS levels < 1	`https://github.com/njh/mqtt-sn-tools/pull/46`	Dec 4, 2020	Merged on Jan 15, 2021

I herewith assure that I wrote the present thesis titled *Software Updates for the Internet of Things: An Extension and Evaluation of the SUIT Implementation in RIOT* independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, July 1, 2021

(Vera Clemens)