Interface Design Description (IDD) for Service Discovery over MQTT

Abstract

This document describes for the Interface Design Description (IDD) of the Arrowhead Service Discovery service's interfaces.

An Interface Design Description provides a detailed description of how the service is implemented/realized by using the Communication Profile and the chosen technologies.

This document outlines interfaces, message formats, metadata, and other important information to be able to use the Service Discovery service and its interfaces.

1. Interface Design Description Overview 4

2. Service Interfaces 4

2.1. Interface 1: Echo 4

2.1.1. Information Model 5

2.1.2. Error handling 5

2.1.3. Interaction with consumers 6

2.2. Interface 2: Query 6

2.2.1. Information Model 6

2.2.2. Error handling 8

2.2.3. Interaction with consumers 8

2.3. Interface 3: Register 9

2.3.1. Information Model 10

2.3.2. Error handling 11

2.4. Interface 4: Unregister 12

2.4.1. Information Model 13

2.4.2. Error handling 13

2.4.3. Interaction with consumers 14

3. Security 14

3.1. Certificates 14

3.2. Payload protection 14

4. References 15

5. Revision history 15

5.1. Amendments 15

5.2. Quality Assurance 15

6. Appendixes 16

1. Interface Design Description Overview

This section contains pointers to Service Description (SD) documents.

Table 1 Pointers to SD documents

Service description	Path
Service Discovery	`tbd`

This document describes how to utilize the Service Registry system's Service Discovery service.

- Protocol: MQTT 3.1.1
- Encoding: JSON
- Compression: none
- Security: Using TLS and X.509 certificates (server and client)

2. Service Interfaces

Since MQTT is based on the Publish/Subscribe communication pattern, it is necessary to encapsulate all messages between and consumer and the Service registry in a top-level message format. This message format is named HTTP-over-MQTT and can encode commonly used information fields from HTTP such as method, content-type, response code, etc.

Each service interface in MQTT is thus based on the combination of a publish/subscribe communication pattern together with an encapsulation message with the true JSON message encapsulated in a payload tag.

In order to emulate request-response, the *replyTo* tag must contain the response topic created by the client system.

Message encapsulation

```
{
 "method": "string",
 "responseCode": “string”
 "replyTo": “string”,
 “payload”: <JSON object>
}
```

2.1. Interface 1: Echo

Below are the specifics of this interface:

- The data model is plain text.
- The true message semantics is the same as the REST-based ServiceRegistry, with the extension that the messages are added as a payload field in an REST-Over-MQTT message
- No ontologies are in use.
- No schemas is currently defined.
- No payload encryption is used. With MQTT 5.0 it will be possible to use payload encryption between different systems.
- The topic is: **ah/serviceregistry/echo**

Table 2 Function description

Function	**Service**	**Method**	**Input**	**Output**
Echo	Service Discovery	GET	-	String

2.1.1. Information Model

The information for Echo is simple. There is no input, and only plain text output.

2.1.2.Parameters

The Echo function does not take any parameters.

2.1.3. Return codes

Only 200 OK are returned.

2.1.4. Error handling

There is no error handling for the Echo function.

2.1.5. Interaction with consumers

Echo only supports read operations, where the response is always a string "Got it". This can be used to test if a system is actually running. No authorization is needed.

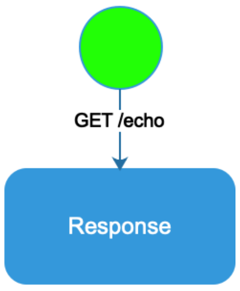

FIGURE 1: ECHO INTERFACE

2.2. Interface 2: Query

Below are the specifics of this interface:

- The data model is JSON.
- The true message semantics is the same as the REST-based ServiceRegistry, with the extension that the messages are added as a payload field in an REST-Over-MQTT message
- No ontologies are in use.
- No schemas are currently defined.
- No payload encryption is used. With MQTT 5.0 it will be possible to use payload encryption between different systems.
- The topic is: **ah/serviceregistry/query**

Table 3 Function description

Function	Service	Method	Input	Output
Query	Service Discovery	POST	ServiceQuery Form	ServiceQueryList

2.2.1. Information Model

In order to get a list of services, a ServiceQueryForm message must be POST:ed to the /query endpoint. The response upon success is a ServiceQueryList.

Input: ServiceQueryForm message

```
{
 "serviceDefinitionRequirement": "string",
 "interfaceRequirements": [
   "string"
 ],
 "securityRequirements": [
   "NOT_SECURE"
 ],
 "metadataRequirements": {
   "additionalProp1": "string",
   "additionalProp2": "string",
   "additionalProp3": "string"
 },
 "versionRequirement": 0,
 "maxVersionRequirement": 0,
 "minVersionRequirement": 0,
 "pingProviders": true
}
```

Output: ServiceQueryList

```
{
 "serviceQueryData": [
   {
     "id": 0,
     "serviceDefinition": {
       "id": 0,
       "serviceDefinition": "string",
       "createdAt": "string",
       "updatedAt": "string"
     },
     "provider": {
       "id": 0,
       "systemName": "string",
       "address": "string",
       "port": 0,
       "authenticationInfo": "string",
       "createdAt": "string",
       "updatedAt": "string"
     },
     "serviceUri": "string",
     "endOfValidity": "string",
     "secure": "NOT_SECURE",
     "metadata": {
       "additionalProp1": "string",
       "additionalProp2": "string",
       "additionalProp3": "string"
     },
     "version": 0,
     "interfaces": [
       {
         "id": 0,
         "interfaceName": "string",
         "createdAt": "string",
         "updatedAt": "string"
       }
     ],
     "createdAt": "string",
     "updatedAt": "string"
    }
 ],
 "unfilteredHits": 0
}
```

2.2.2. Parameters

Table 4 Input parameters

Name	Comment	Mandatory
serviceDefinitionRequirement	Name of the required Service Definition	Yes
interfaceRequirements	List of required interfaces	No
securityRequirements	List of required security settings	No
metadataRequirements	Key value pairs of required metadata	No
versionRequirement	Required version number	No
maxVersionRequirement	Maximum version requirement	No
minVersionRequirement	Minimum version requirement	No
pingProviders	Return only available providers	No

2.2.3.Return codes

Table 5 Return codes

Code	Name	Comment
200	OK	Normal response, no error
400	Bad request	Incorrect request
401	Not authorized	Missing or faulty credentials
500	Internal server error	Database failure, hardware error etc.

2.2.4. Error handling

If the request was successful, a ServiceQueryList is returned with a response code of 200. If an error occurs, for example due to a incorrectly formatted request, an error message is returned with the reason.

2.2.5. Interaction with consumers

Figure 2 shows how a client must perform a query.

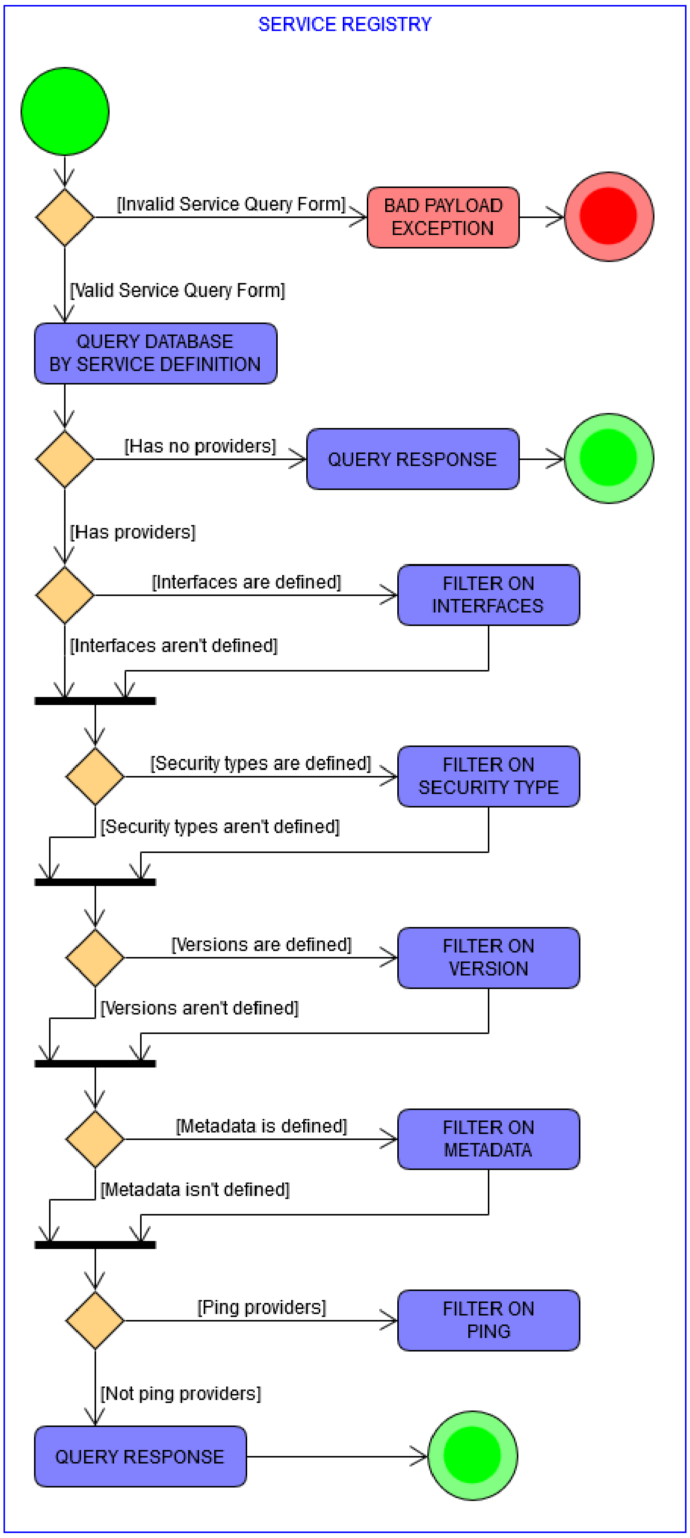

FIGURE 2: QUERY OPERATION

2.3. Interface 3: Register

Below are the specifics of this interface:

- The data model is JSON.
- The true message semantics is the same as the REST-based ServiceRegistry, with the extension that the messages are added as a payload field in an REST-Over-MQTT message
- No ontologies are in use.
- No schemas is currently defined.
- No payload encryption is used. With MQTT 5.0 it will be possible to use payload encryption between different systems.
- The topic is **ah/serviceregistry/register**

Table 6 Function description

Function	Service	Method	Input	Output
Register	Service Discovery	POST	ServiceRegistry Entry	ServiceRegistryEntry

2.3.1. Information Model

In order to register a service, a ServiceRegistryEntry message must be POST:ed to the /register endpoint. The response upon success is an updated ServiceRegistryEntry message with all fields filled in.

Below is the input information model.

Input: ServiceRegistryEntry message

```
{
  "serviceDefinition": "string",
  "providerSystem": {
    "systemName": "string",
    "address": "string",
    "port": 0,
    "authenticationInfo": "string"
  },
  "serviceUri": "string",
  "endOfValidity": "string",
  "secure": "NOT_SECURE",
  "metadata": {
    "additionalProp1": "string",
    "additionalProp2": "string",
    "additionalProp3": "string"
  },
  "version": 0,
  "interfaces": [
    "string"
  ]
}
```

Below is the output information model.

Output: ServiceRegistryEntry message

```
{
  "id": 0,
  "serviceDefinition": {
    "id": 0,
    "serviceDefinition": "string",
    "createdAt": "string",
    "updatedAt": "string"
  },
  "provider": {
    "id": 0,
    "systemName": "string",
    "address": "string",
    "port": 0,
    "authenticationInfo": "string",
    "createdAt": "string",
    "updatedAt": "string"
  },
  "serviceUri": "string",
  "endOfValidity": "string",
  "secure": "NOT_SECURE",
  "metadata": {
    "additionalProp1": "string",
    "additionalProp2": "string",
    "additionalProp3": "string"
  },
  "version": 0,
  "interfaces": [
    {
      "id": 0,
      "interfaceName": "string",
      "createdAt": "string",
      "updatedAt": "string"
 }
 ],
 "createdAt": "string",
 "updatedAt": "string"
}
```

2.3.2. Input parameters

Table 7 shows the input parameters, sent in the request body payload in JSON.

Table 7 Input parameters

Name	Comment	Mandatory
serviceDefinition	Normal response, no error	Yes
providerSystem	Incorrect request	Yes
serviceUri	Missing or faulty credentials	Yes
endOfValidity	Database failure, hardware error etc.	No
secure	Security info	No
metadata	Metadata	No

version	Version of the Service	No
interfaces	List of the interfaces the Service supports	Yes

2.3.3. Return codes

Table 8 Return codes

Code	Name	Comment
201	Created	Normal response, no error
400	Bad request	Incorrect request
401	Not authorized	Missing or faulty credentials
500	Internal server error	Database failure, hardware error etc.

2.3.4. Error handling

If the request was successful, a ServiceRegistryEntry message is returned inside a REST-over-MQTT encapsulation message, with a response code of 200. If an error occurs, for example due to an incorrectly formatted request, an error message is returned with the reason.

2.4. Interface 4: Unregister

Below are the specifics of this interface:

- The data model is JSON.
- The true message semantics is the same as the REST-based ServiceRegistry, with the extension that the messages are added as a payload field in an REST-Over-MQTT message
- No ontologies are in use.
- No schemas is currently defined.
- No payload encryption is used. With MQTT 5.0 it will be possible to use payload encryption between different systems.
- The topic is **ah/serviceregistry/unregister**

Table 9 Function description

Function	Service	Method	Input	Output

Unregister	Service Discovery	POST	System Name, Address and Port in query parameters	Status

2.4.1. Information Model

The Unregister interface takes its input as query parameter. The response is a status code, where 200 indicates success.

2.4.2. Input parameters

Table 10 shows the input parameters, sent as query parameters.

Table 10 Input query parameters

Name	Comment	Mandatory
service_definition	Name of the service to be removed	Yes
system_name	Name of the Provider	Yes
address	Address of the Provider	Yes
port	Port of the Provider	Yes

2.4.3. Return codes

Table 11 Return codes

Code	Name	Comment
200	OK	Normal response, no error
400	Bad request	Incorrect request
401	Not authorized	Missing or faulty credentials
500	Internal server error	Database failure, hardware error etc.

2.4.4. Error handling

If an error occurs, the response will indicate what the error is.

2.4.5. Interaction with consumers

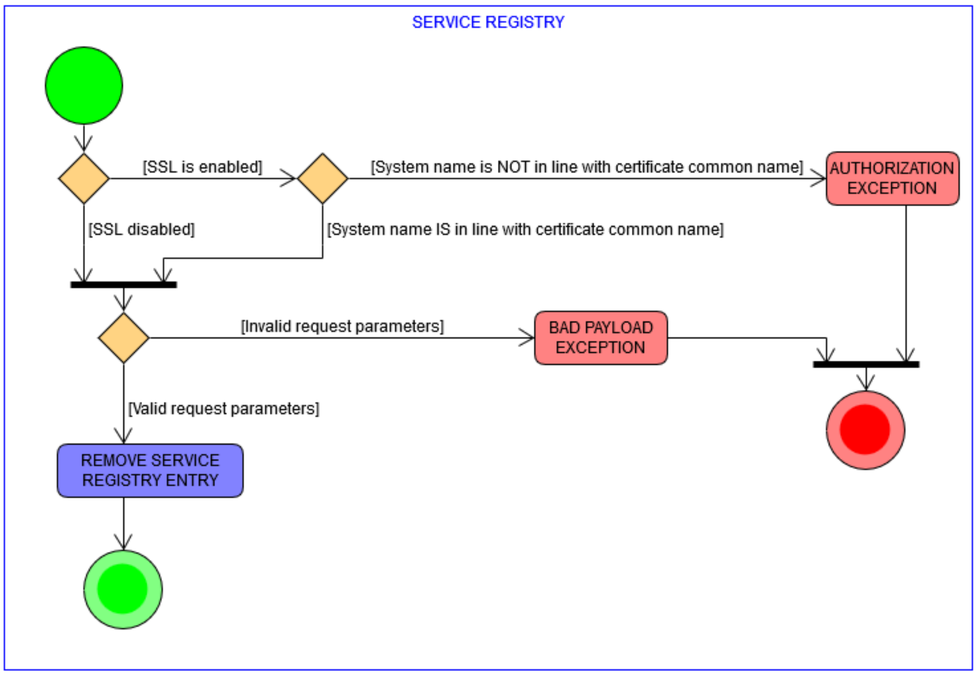
SERVICE REGISTRY
[SSL is enabled]
[System name is NOT in line with certificate common name]
AUTHORIZATION EXCEPTION
[SSL disabled]
[System name IS in line with certificate common name]
[Invalid request parameters]
BAD PAYLOAD EXCEPTION
[Valid request parameters]
REMOVE SERVICE REGISTRY ENTRY

FIGURE 3: UNREGISTER OPERATION

3. Security

This service can either run unencrypted over MQTT, or using TLS plus server and client side X509 certificates. MQTT can also use username and password login management. An MQTT broker can also be configured for client-level access control. This feature cannot today be used by an Arrowhead local cloud, but it should be possible to add features to an MQTT broker so that Arrowhead Authorization is linked to Access control lists (ACL) in the MQTT domain.

3.1. Certificates

This IDD is using the same certificates as provided byt eh Java Spring versions. The only difference is that some MQTT libraries, such as PAHO, and brokers, such as Mosquitto, only supports PEM-encoded files. The PKCS#12 certificates must therefore be converted into .pem / .crt files by a tool, such as openssl.

3.2. Payload protection

Currently, this IDD can run directly over unencrypted TCP or encrypted TLS. Due to the nature of the Publish/Subscribe nature of MQTT, it is not possible with version 3.1.1 to handle a per client encryption. MQTT version 5.0 do support this feature, but today's open source libraries do not support version 5.0 yet.

4. References

[1] MQTT spec. 3.1.1 ISO /IEC 20922:2016, URL https://www.iso.org/standard/69466.html

[2] MQTT version 5.0, Banks, Briggs, et al. OASIS 2019.Message Queue Telemetry Transport. URL http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[3] Eclipse Mosquitto, URL https://mosquitto.org/
[4] Eclipse PAHO MQTT library, URL https://www.eclipse.org/paho/

5. Revision history

5.1. Amendments

No.	Date	Version	Subject of Amendments	Author
1	2015-02-15	1.0	Revision of text	Michele Albano / Luis Ferreira
2	2015-09-30	1.1	Refinement of the structure	Michele Albano / Luis Ferreira
3	2020-06-07	2.0	Major update	Jerker Delsing
4	2020-06-17	2.1	Added MQTT text	Jens Eliasson
5	2020-06-18	2.2	Added pictures, appendix etc	Jens Eliasson
6	2020-08-12	2.3	Added response codes	Jens Eliasson
7	2020-08-15	2.4	Text cleanup, added parameters	Jens Eliasson

5.2. Quality Assurance

No.	Date	Version	Approved by
1			
2			

6. Appendixes

Appendix A: MQTT Communication profile (CP)

Document title
Communication Profile – MQTT

Date
2020-06-18

Author
Jens Eliasson

Contact
jens.eliasson@thingwave.eu, +46 70-222 4074

Document type
Communication Profile

Version
1.0

Status
DRAFT

Page
1 (12)

Communication Profile – MQTT

Abstract
This profile describes the use of the MQTT (Message Queue Telemetry Transport) protocol for data communication within an Arrowhead local cloud.

ARTEMIS Innovation Pilot Project: Arrowhead
THEME [SP1-JTI-ARTEMIS-2012-AIPP4 SP1-JTI-ARTEMIS-2012-AIPP6]
[Production and Energy System Automation Intelligent-Built environment and urban infrastructure for sustainable and friendly cities]
Grant agreement no: 332987. Project Coordinator: Professor Jerker Delsing | Luleå University of Technology

Table of contents

2.1. Communication Profile Identifier 4
2.2. Open-source libraries 4
2.3. Brokers 4
3.1. Publish-Subscribe 5
3.2. Request-Response 5
3.3. Endpoint Description 5
7.1. Operations 9
7.2. Data 9
8.1. Message Queue Telemetry Transport (MQTT) 10
9.1. Amendments 12
9.2. Quality Assurance 12

1. Abbreviations

MQTT	Message Queue Telemetry Transport
CP	Communication Profile
HTTP	Hypertext Transfer Protocol
M2M	Machine-to-Machine
REST	Representational State Transfer
XML	Extensible Markup Language
EXI	Efficient XML Interchange
CRUD	Create/Read/Update/Delete
TLS	Transport Layer Security
JSON	JavaScript Object Notation
CBOR	Concise Binary Object Representation

2. Introduction

MQTT [1], [5] is a specialized web transfer protocol for use with constrained nodes and constrained networks. One of the main goals of MQTT is to is to simplify data distribution and to disconnect a data producer and the consumers of data. For this purpose, a message broker is used to as act as a middle hand. A publishes sends its messages to the broker, and the broker in turn distribute the message to all subscribers that has subscribed to a specific topic.

Data communication with MQTT can be over TCP or Websockets. Data can be encrypted using TLS with or without client X509 certificates.

MQTT was originally developed by IBM, but is today managed by OASIS.

2.1. Communication Profile Identifier

This communication profile is identified as **MQTT**. Currently only MQTT 3.1.1 s supported by the Arrowhead Framework. Support for MQTT 5.0 could enable some very interesting features, such as per client encryption.

2.2. Open-source libraries

The most widely used open-source library of MQTT is the PAHO library [4] from Eclipse. PAHO supports both Java, Python, JavaScript and Golang. There are also many other open-source implementations such as Hive and Mosquitto.

2.3. Brokers

As of this writing, the Mosquitto broker [6] from Eclipse is commonly used. Mosquitto is written in C and comes with test applications and a library for C-based applications.

3. Message Exchange Patterns

This chapter explains how to implement a common set of message exchange patterns using this communication profile.

3.1. Publish-Subscribe

The core specification of MQTT [1] is based on the publish-subscribe message exchange pattern. There are no clients or servers, only publishers and subscribers.

3.2. Request-Response

A request-response based communication protocol, i.e. HTTP, is widely used for many different applications. It is therefore important to be able to emulate request-response over MQTT. The Arrowhead Framework therefore provides an emulation layer, where a basic subset of HTTP is transported over MQTT. The use of dynamic response topics where only a specific publisher is subscribing to, and sending that topic name in the request so that a subscriber can respond to the dynamic topic.

Using this request-response message exchange pattern, it is possible to implement typical request-response patterns such as CRUD operations (GET/POST/PUT/DELETE), and thereby performing service registration etc.

3.3. Endpoint Description

An endpoint utilizing this communication profile must expose the following information to its communicating parties.

IP	Broker IP
Port	Broker port
SystemName	The name of the system
Service	Topic name for a service (ex: “sensor/temperature”)

This information will form an URL valid for MQTT clients:

mqtt://<IP>:<Port>/<SystemName>/<Service>

Broker local clouds that only allow exactly one and only one broker, the short URL form below can be used:

mqtt://<SystemName>/<Service>

4. Security

The core specification of MQTT [1] defines a mapping to TLS. MQTT can run with broker certificate only, and with mandatory client certificates. MQTT also supports additional security with usernames and passwords. A broker can also be configured to only allow specific clients to access certain topics.

5. Metadata

The required additional metadata related to this communication profile is the URI of the MQTT broker (IP address/name, port), and security settings (username, password, certificate, certificate password). Topic prefix is another settings, i.e. should all topics in a local cloud be prefixed with a certain name, such as the local cloud name or “ah”.

6. Data Format

This communication profile supports any binding to data formats such as plain text, XML, JSON, CBOR [3], etc. but SenML using plain text JSON is the mostly used format for sensor and actuator data messages. The HTTP-over-MQTT emulation layer is based on currently based on JSON, but CBOR and XML could be added if needed.

7. Description Format

This chapter describes which documentation artifacts that is required for the description of a service utilizing this communication profile.

7.1. Operations

The operations of a service using this communication profile should be documented like below:

Operation	Service	Method	Input	Output

Here is an example:

getTemp	ah/tempSys1/temp	Subscribe (GET)	-	TempData
setTemp	ah/tempSys1/temp	Publish (POST)	TempData	-

7.2. Data

The data, either input or output should be described using the JSON format, but other formats can be used as well.

8. Standards and Demarcations

Specification	**Type**	**Version**
Message Queue Telemetry Transport (MQTT)	Oasis standard	3.1.1
Message Queue Telemetry Transport (MQTT)	Oasis standard	5.0

8.1. Message Queue Telemetry Transport (MQTT)

No demarcations are in affect for version 3.1.1. MQTT 5.0 is currently not supported.

References

[1] Banks, Briggs, et al. OASIS 2019.Message Queue Telemetry Transport.
URL http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[2] Jenning, Shelby et al. SenML standards, URL https://tools.ietf.org/html/rfc8428

[3] Bormann, Hoffman, RFC 7049. URL https://tools.ietf.org/html/rfc7049

[4] Eclipse PAHO MQTT library, URL https://www.eclipse.org/paho/

[5] MQTT spec. 3.1.1 ISO /IEC 20922:2016
URL https://www.iso.org/standard/69466.html

[6] Eclipse Mosquitto, URL https://mosquitto.org/

9. Revision history

9.1. Amendments

No.	Date	Version	Subject of Amendments	Author
1	2013-10-28	0.5	Document created	Markus Klisics
2	2013-11-26	0.6	Updated to cover CoAP only	Jens Eliasson
3	2020-06-15	0.9	Updated to Arrowhead 4.1.3.	Jens Eliasson
4	2020-06-16	1.0	Text updates, etc	Jens Eliasson

9.2. Quality Assurance

No.	Date	Version	Approved by
1	2013-12-13	0.6	Jens Eliasson
2			