

Interface Design Description (IDD) for Orchestrator over MQTT

Abstract

This document describes for the Interface Design Description (IDD) of the Arrowhead Orchestrator 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 Orchestrator service and its in

1. Interface Design Description Overview	3
2. Service Interfaces	3
2.1. Interface 1: Echo	4
2.1.1. Information Model	4
2.1.2. Error handling	4
2.1.3. Interaction with consumers	5
2.2. Interface 2: Orchestration	5
2.2.1. Information Model	5
2.2.2. Error handling	7
2.2.3. Interaction with consumers	7
2.3. Interface 3: Start store Orchestration by ID	8
2.3.1. Information Model	9
2.3.2. Error handling	10
3. Security	10
3.1. Certificates	10
3.2. Payload protection	10
4. References	11
5. Revision history	11
5.1. Amendments	11
5.2. Quality Assurance	11
6. Appendixes	12

1. Interface Design Description Overview

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

Table 1 Pointers to SD documents

Service description	Path
Orchestration service	tbd

This document describes how to utilize the Orchestrator system's Orchestration service. Other services, such as the Management service, is not provided over MQTT.

- 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 Orchestrator, 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.

Table 4 Function description

Function	Service	Method	Input	Output
Echo	Service Discovery	GET	-	String

2.1.1. Information Model

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

2.1.2. Error handling

There is no error handling for the Echo interface.

2.1.3. 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.

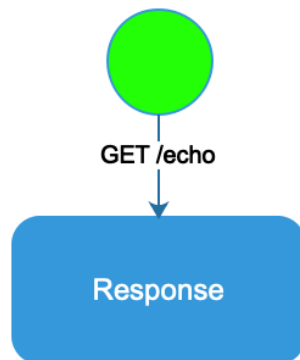


FIGURE 1: ECHO INTERFACE

2.2. Interface 2: Orchestration

Below are the specifics of this interface:

- The data model is JSON.
- The true message semantics is the same as the REST-based Orchestrator, 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.

Table 4 Function description

Function	Service	Method	Input	Output
Orchestration	Orchestration	POST	ServiceRequestForm	Orchestration Response

2.2.1. Information Model

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

Input: ServiceRequestForm message

```
{
  "requesterSystem": {
    "systemName": "string",
    "address": "string",
    "port": 0,
    "authenticationInfo": "string"
  },
  "requestedService": {
    "serviceDefinitionRequirement": "string",
    "interfaceRequirements": [
      "string"
    ],
    "securityRequirements": [
      "NOT_SECURE", "CERTIFICATE", "TOKEN"
    ],
    "metadataRequirements": {
      "additionalProp1": "string",
      "additionalProp2": "string",
      "additionalProp3": "string"
    },
    "versionRequirement": 0,
    "maxVersionRequirement": 0,
    "minVersionRequirement": 0
  },
  "preferredProviders": [
    {
      "providerCloud": {
        "operator": "string",
        "name": "string"
      },
      "providerSystem": {
        "systemName": "string",
        "address": "string",
        "port": 0
      }
    }
  ],
  "orchestrationFlags": {
    "additionalProp1": true,
    "additionalProp2": true,
    "additionalProp3": true
  }
}
```

Output: Orchestration Response

```
{
  "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. Error handling

If the request was successful, a **Orchestration Response** is returned 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.

2.2.3. Interaction with consumers

Figure 2 shows how a client must perform an orchestration operation.

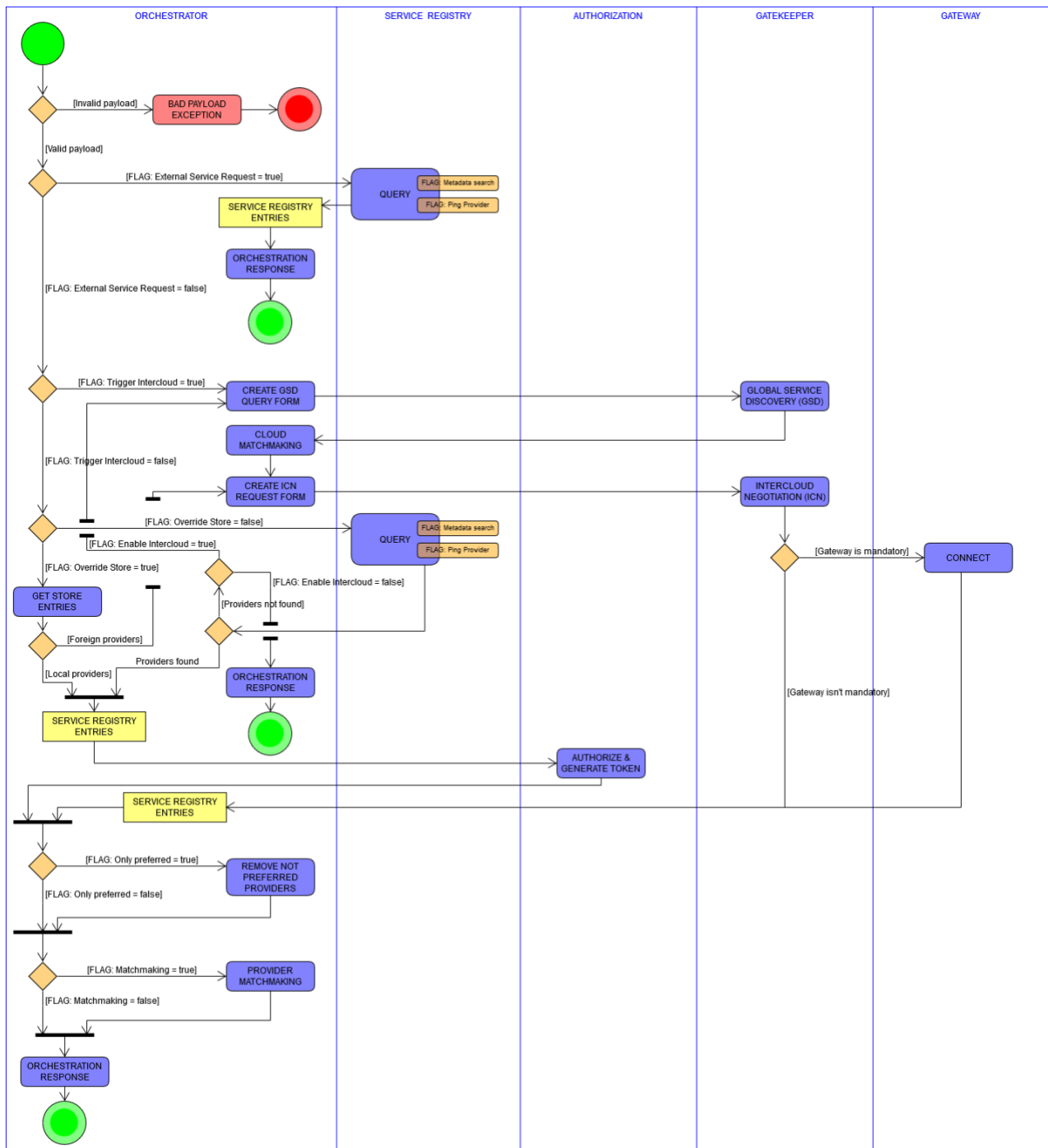


FIGURE 2: ORCHESTRATION OPERATION

2.3. Interface 3: Start store Orchestration by ID

Below are the specifics of this interface:

- The data model is JSON.
- The true message semantics is the same as the REST-based Orchestrator, 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.

Table 5 Function description

Function	Service	Method	Input	Output
Start store Orchestration by ID	Service Discovery	GET	id	Orchestration Response

2.3.1.Information Model

In order to start an orchestration with a known id, a client can perform a GET to /orchestrator/orchestration/{id}. The id is sent as a query parameter The response upon success is an updated ServiceRegistryEntry message with all fields filled in.

Output: Orchestration Response message

```
{
  "response": [
    {
      "provider": {
        "id": 0,
        "systemName": "string",
        "address": "string",
        "port": 0,
        "authenticationInfo": "string",
        "createdAt": "string",
        "updatedAt": "string"
      },
      "service": {
        "id": 0,
        "serviceDefinition": "string",
        "createdAt": "string",
        "updatedAt": "string"
      },
      "serviceUri": "string",
      "secure": "TOKEN",
      "metadata": {
        "additionalProp1": "string",
        "additionalProp2": "string",

```

```

        "additionalProp3": "string"
    },
    "interfaces": [
        {
            "id": 0,
            "createdAt": "string",
            "interfaceName": "string",
            "updatedAt": "string"
        }
    ],
    "version": 0,
    "authorizationTokens": {
        "interfaceName1": "token1",
        "interfaceName2": "token2"
    },
    "warnings": [
        "FROM_OTHER_CLOUD", "TTL_UNKNOWN"
    ]
}
]
}

```

2.3.2. Error handling

If the request was successful, an Orchestration Response 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.

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 by the 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, e.g. 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 Orch. text	Jens Eliasson
5	2020-06-23	2.2	Added pictures, appendix etc	Jens Eliasson
6				

5.2. Quality Assurance

No.	Date	Version	Approved by
1			
2			

6. Appendixes

Appendix A: MQTT Communication profile (CP)

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.

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 simplify data distribution and to disconnect a data producer and the consumers of data. For this purpose, a message broker is used to act as a middle hand. A publisher sends its messages to the broker, and the broker in turn distributes the message to all subscribers that have 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 is 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] Jennings, 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			