

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 methods.

1. Interface Design Description Overview 3

2. Service Interfaces 3

2.1. Service 1: **Echo** 3

2.1.1. Information Model 4

2.1.2. Error handling 4

2.1.3. Interaction with consumers 4

2.2. Service 2: **Orchestration** 4

2.2.1. Information Model 5

2.2.2. Parameters 6

2.2.3. Response codes 6

2.2.4. Error handling 7

2.2.5. Interaction with consumers 7

2.3. Service 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 10

5. Revision history 11

5.1. Amendments 11

5.2. Quality Assurance 11

6. Appendixes 11

## Interface Design Description Overview

This document describes how to utilize the Orchestrator system’s Echo and Orchestration services. Other services, such as the Management service, are not provided over MQTT.

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

## 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>

}

The Orchestrator, being a part of the Arrowhead Framework [1], provides three services over MQTT; **Echo**, **Orchestration**, and **Start store Orchestration by ID**. The Echo service is used to check the availability of the Orchestrator system. The Orchestration service provides a way for a system to automatically know which services to consume. The Start store Orchestration by ID service can be used if the consumer knows its' ID, and can used this for store-based orchestration when the service returns the top priority local provider of all services contained by the orchestrator store database for the requester system.

## Service 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 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** |
| Echo | Service Discovery | GET | - | String |

## Information Model

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

## Error handling

There is no error handling for the Echo interface.

## 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

## Service 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 |

## 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

}

## Parameters

This interface does not take any query path parameters.

## Response codes

|  |  |  |
| --- | --- | --- |
| Code | **Meaning** | **Comment** |
| 200 | Successful request | Success |
| 400 | Bad request | If an incorrect parameter is used |
| 401 | Unauthorized | Access denied |
| 500 | Internal server error | In case of database errors etc. |

## 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.

## Interaction with consumers

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

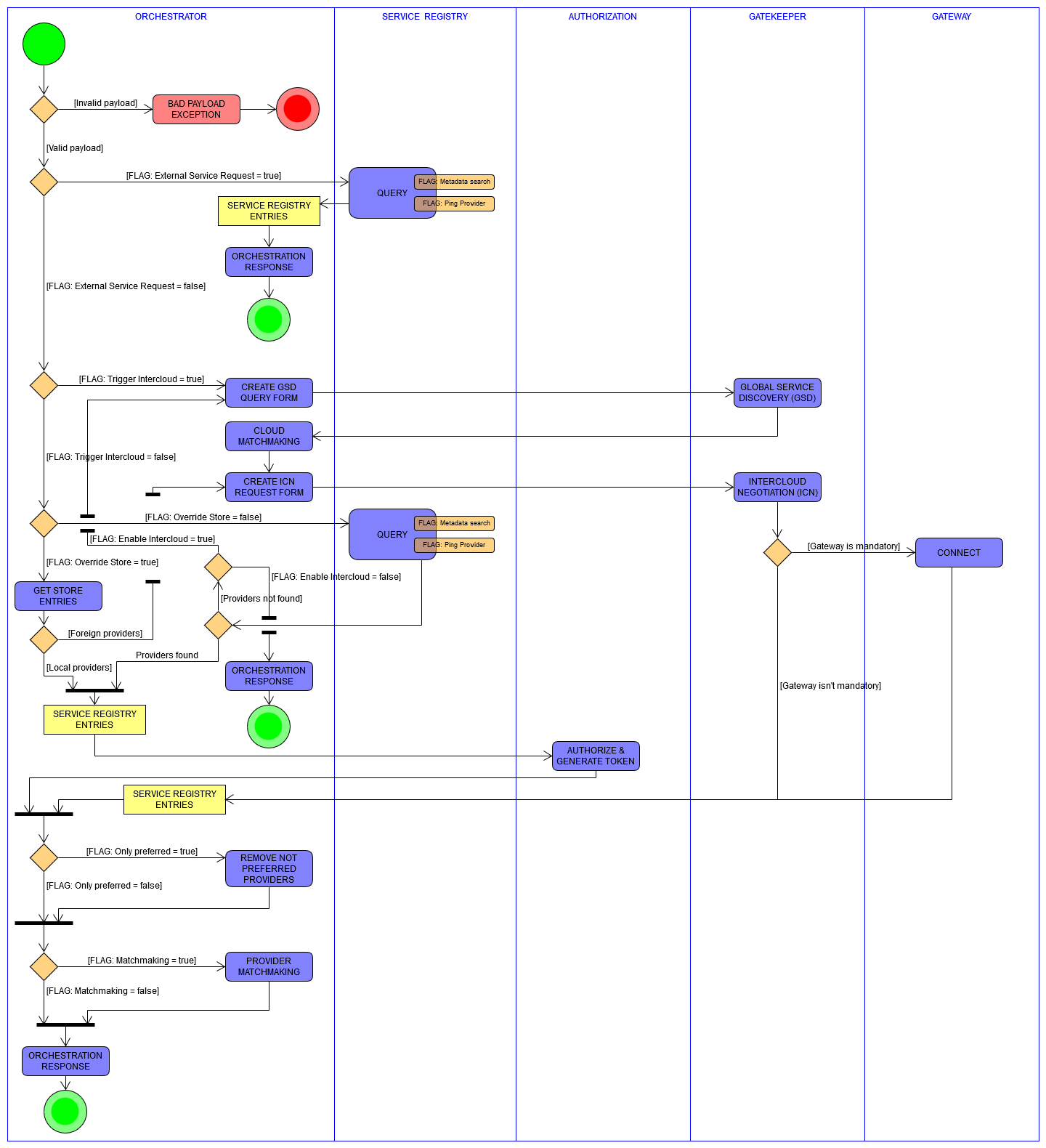


Figure 2: Orchestration operation

## Service 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 are 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 |

## 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"

]

}

]

}

## 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.

## Security

This IDD 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.

## 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 [5].

## 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 most of today’s open source libraries do not support version 5.0 yet.

## 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. OpenSSL, <https://www.openssl.org/>

## Revision history

## 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 | 2020-07-06 | 2.3 | Harmonized with other IDD docs | Jens Eliasson |
| 7 |  |  |  |  |

## Quality Assurance

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Date | Version | Approved by |
| 1 |  |  |  |
| 2 |  |  |  |

## Appendixes

Appendix A: MQTT Communication profile (CP)