Orchestration Service Description

**Abstract**

This document defines the Orchestration Service that supports - among others - inter-cloud interactions.

This Orchestration Service provides an appropriate ”choreography” for a requested functionality by telling the requester what and how to consume.

This Service is a composed one (compared to atomic ones), meaning that it combines several Services together (that from i.e. Service Registry, Authorization, etc.) in order to provide a consistent state within the Local Cloud.

Application Systems should only be able to start a servicing connection with each other, once they have received orchestration.

1. Orchestration Service Overview

This document describes an Arrowhead service, including its interfaces, functions and information model. It is a Core Service from the System Management group.

The Orchestration Service targets the same end goal as the OrchestrationStore Service from previous generations. However, the scope and functionality is extended, and the reliance on an Store is just a part of the internal workings.

In this current release, orchestration push is indirectly supported. There is no dedicated Service that needs to be implemented in an Application System, as previous generations.

In here, this architecture relies on the use of Event Handler to notify the affected Systems that their orchestration status needs to change. Therefore, only the Event Subscribe Service needs to be implemented in the Systems, and they need to be able to listen and act on a specific renew-orchestration-configuration message. The implementation of this is use case specific, not general.

This pull-typed service utilizes other Core Systems to allocate one or more suitable Service Providers to the request.:

* Orchestration Store: in certain cases, hardwired orchestration rules have to be realized
* Service Registry: fetching online Providers that are currently active and offer the exact service interface with appropriate metadata description
* Authorization: checking whether the Consumer at hand is authorized to consume from any of the Service Providers
* Gatekeeper: if inter-Cloud servicing is enabled or triggered, the Gatekeepers handle the Service lookup and negotiations processes with other Clouds
* Other Core Systems: if needed, the Orchestrator has to consult (e.g. resource availability and reservation with the QoS Manager – currently not implemented)

***Note****: This Service might return different results if re-queried instantly since the status of the SoS might change in that time. Therefore, a requested orchestration cannot be re-accessed after receiving it.*

Still, this complex process is hidden from the requester Consumer, but three entities can affect what will be the result:

* The Cloud operator can make Orchestration Store entries that will enforce design time hardwired connections during runtime
* The potential Service Provider: it can post special metadata flags in the Service Registry that will alter the orchestration process (e.g. when it marks that to access its Service, an authorization token is required).
* The Service Consumer can alter the process via setting special flags in the request.

As discussed in [1], this Service must serve a multitude of devices with various capabilities and operational restrictions. To this end, the Service Request Form (SRF) now includes various fields to tailor the orchestration process.

1. Abstract Interfaces

The purpose of this section is to clarify the interface involved in the Orchestration Service. There is only one request-typed Service offered publicly towards Application Systems.

# Service Request



**Figure 1. The Orchestration process overview**

1. Abstract Information Model

## Common Arrowhead Classes

***Reminder:*** The System and Service hierarchy elements are represented in three common Arrowhead classes that describe Arrowhead Systems (within their Cloud), Arrowhead Services (that describe the desired functionality) and the Arrowhead Clouds (that help identifying other Local Clouds from our own). These classes are commonly used among all G3 proof-of-concept implementation (as seen on Figure 2).



**Figure 2. The common Arrowhead Classes**

## Service Request Form

A request is called *ServiceRequestForm*. It contains the following information, as presented in Table 1. This is the payload that needs to be sent when the Application System wishes to consume a Service or generally tries to access its default orchestration information.

Table 1 ServiceRequestForm type description

|  |  |
| --- | --- |
| **Field** | **Description** |
| RequesterSystem: ArrowheadSystem | The ArrowheadSystem identifier that is requesting the orchestration. Mandatory field. |
| RequesterCloud: ArrowheadCloud | Not to be used by Application Systems. This field is only used when there is an inbound request from another Cloud. This can only be submitted by the Gatekeeper. |
| RequestedService: ArrowheadService | Not mandatory. When not filled out, the Orchestrator operate in the ”default configuration” mode. When filled out, other fields will affect how the orchestration will work – but the Service will be specified by the Consumer in advance then. |
| OrchestrationFlags: HashMap | These flags can be set to tailor the orchestration process. |
| PreferredProviders | There can be a list from the Consumer stating which other Application Systems (even from other Clouds) it wishes to connect. If this field is filled out, the Orchestrator will try to facilitate these preferences. |
| RequestedQoS | This object will be presenting the Consumer’s selected QoS level. Not implemented. |

## Orchestration Form

The response to a Service Request is the Orchestration Response, which can contain several Orchestration Forms.

Table 2 OrchestrationForm type description

|  |  |
| --- | --- |
| **Field** | **Description** |
| Service: ArrowheadService | The exact Service that needs to be consumed. This response matches the ServiceRequestForm’s (if that was filled out), but it comes from the exact Service instance with proper metadata and interface identifier (only one). |
| Provider: ArrowheadSystem | The Application System description that will provide the Service. |
| ServiceURI: String | The base URL within the Service Provider that will contain the seeked Service. |
| Instruction: String | Optional additional data from the orchestration. |
| AuthorizationToken: String | The encrypted authorization token. |
| Signature: String | The signature from the Authorization System for the token (verifying the token issuing). |
| Warnings: List<OrchestratorWarnings> | Enumeration with values that signal certain things, such as the provider is from another Cloud, or the service offering might have expired already. |

1. Internal Workings

Through setting the SRF flags, the following orchestrations can be requested (Fig. 3).



**Figure 3. Overview of the orchestration process alternatives**

1. Default configuration

For certain cases, where App. Systems don’t have operational autonomy or lack processing power, a default set of to-be-consumed Services and their pre-set Providers can be stored. Upon request, these Systems would not get a matchmaker decision, but can fetch this hardwired configuration through the same interface. This happens, when the SRF doesn’t specify a Service.

To this end, the Orchestration Form (part of the Response) contains an “Instructions” field that is currently only used by this scenario to pass on configurational data. Since other fields of the Orchestration Response contain the Provider’s address and the exact type of the Service interface, this field can be used to store flags, command arguments, etc.

1. Dynamical orchestration

This is the default methodology in this generation. Here, the Orchestrator is a matchmaker, that dynamically tries to allocate Service Providers to the request. During this, other Core Systems are involved:

* Fetching a list of suitable on-line Service Providers from the Service Registry that offer the Service at hand,
* Filtering this list of possible Service Providers based on the authorization status of the Service Consumer;
* Further ﬁltering this list to ﬁnd out which Service Provider(s) can satisfy the QoS expectations – currently not implemented
* Choosing one Service Provider from this ﬁltered list (matchmaking);
* Then making the QoS resource reservations for this optimal (Provider - Service - Consumer) combination – currently not implemented
* If the request can’t be satisfied locally, then the Gatekeeper’s services are used to extend the Service discovery to other Clouds;
* Negotiating the terms with a suitable Arrowhead Cloud (inter-Cloud orchestration)
* Responding to the requester System with an Orchestration Response message

***Note****: Inter-Cloud orchestration is detailed in [2]. From the Orchestrator’s point of view, it only requires the usage of two Core Services from the Gatekeeper, see [3].*

* 1. Matchmaking

There are various possibilities for matching a request with a service within the local cloud. This process is called intra cloud matchmaking and currently there are two matchmaking algorithms implemented: a default and a random.

The default intra cloud matchmaking algorithm takes a list of available services (as service registry responses) and a list of preferred providers. If no preferred provider is specified or none of the preferred providers are available, the first available provider is returned. Otherwise, the first provider is returned which is both available and preferred.

The random intra cloud matchmaking algorithm works similarly to the default, but when no preferred provider is specified, or none of the preferred providers are available, a random available one is returned.

A future plan is to implement a more sophisticated intra cloud matchmaking algorithm based on the service security type (included in the service registry response). The service security type can have 3 levels: “not secure”, “certificate-based” or “token-based”. Certificate-based is more secure than not secure, and token-based is more secure than certificate-based. The algorithm should be able to specify a minimum required security level and should work as follows. If there is any preferred provider which meets the minimum security level, the first one should be returned. If no preferred provider is above the minimum level, but there is a non-preferred above the level, the first one should be returned.

1. Orchestration based on Store

In certain cases, Application Systems request specific Services they wish to consume (e.g. method invocation). Therefore, they turn to the Orchestrator to point them to a Service Provider. However, certain restrictions might apply to certain Consumers: they can only consume that particular Service from a pre-set list of Providers that follow each other in a priority order (primary Provider might have several backups).

However, these Providers might not all be online at the time of the request, therefore the Orchestrator has to iterate through this list and check which entry is valid.

Moreover, Store entries can also point to Service Providers that are not within the Local Cloud, therefore inter-Cloud orchestration is required.

***Note:*** *For the Orchestration Store’s architecture, database structure and management interfaces, please refer to [4].*

1. Inbound inter-Cloud servicing requests

So far, only the initialization of inter-Cloud servicing was described. However, an Orchestrator can also be the subject of an inbound request from another Cloud. Here, the local Gatekeeper represents the other Cloud’s request by submitting a specialized SRF (marked with externalRequest flag). This SRF is served differently, and the orchestration response is passed back to another Cloud through the Gatekeepers, as seen on [3].

1. References

[1] Arrowhead G4.0 generic SoSDD - Core System architecture

[2] Arrowhead Gatekeeper G4.0 SysDD

[3] Dynamical Orchestration Process – Sequence Diagram (PDF)

[4] Arrowhead Orchestrator G4.0 SysDD

1. Revision history

# Amendments

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Date | Version | Subject of Amendments | Author |
| 1 | 2016-08-23 | 0.1 | Initial version | Csaba Hegedűs |
| 2 | 2018-02-19 | M3 | Updated document to M3 | Csaba Hegedűs |
| 3 | 2018-05-23 | G4.0 | Updated to G4.0 | Zoltán Umlauf |

# Quality Assurance

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