Orchestrator System SysDD

**Abstract**

This document provides an overview on the Orchestrator System within Arrowhead generation 4.0.

1. System Design Description Overview

The Orchestrator provides runtime (late) binding between Application Systems.

The primary purpose for the Orchestrator System is to provide Application Systems with orchestration information: where they need to connect to. The outcome of the “*Orchestration Service*” include rules that will tell the Application System what Service provider System(s) it should connect to and how (acting as a Service Consumer). Such orchestration rules include:

* Accessibility information details of a Service provider (e.g network address and port),
* Details of the Service instance within the provider System (e.g. base URL, IDD specification and other metadata),
* item Authorization-related information (e.g. access token and signature),
* Additional information that is necessary for establishing connection.

This orchestration rule information can reach the given Application System (consumer) in two different ways: the System itself can request it ("pull") or the Orchestrator itself can update the System when it is needed ("push method"). However, in both cases, there shall be an underlying, hidden process ("*orchestration process*"), which ensures the consistence of state between the various Core Systems.

In G4.0, only the pull method is implemented and the Orchestrator shall negotiate with the other Core Systems while trying to facilitate the new service request (or trying to push a new status). This is necessary for the following cases and requirements (basically, when ad hoc, unsupervised connections are not allowed):

* When accountability is required for all Systems in the Local Cloud: connections cannot be established without the knowledge, approval and logged orchestration events of the Core Systems ("central governance").
* QoS and resource management reasons: ad hoc peer-to-peer connections cannot be allowed in certain managed networks and deployment scenarios. Every connection attempt shall be properly authorized and its QoS expectations (resource reservations) handled.
* Inter-Cloud orchestration can only happen via negotiations between the two Core System sets. Ad hoc inter-cloud connections shall not be allowed in the Arrowhead framework.

In these cases, when the Orchestrator is the sole entry point to establishing new connections within the Local Cloud, Application Systems do not have the possibility to skip any of the control loops with all the appropriate Core Systems. When such security and safety concerns are not present, the orchestration process might be cut back or these interactions between Core Systems might be limited. Within G4.0, this is not the primary use case, but it is allowed. With the proper self-implemented (modified) and a self-compiled Orchestrator can fit the deployment best.

Therefore, the Orchestrator provides two core Services and may consume many other ones, but at least two -- again, depending on its deployment. Figure 1. depicts the mandatory and optional interfaces of this System.

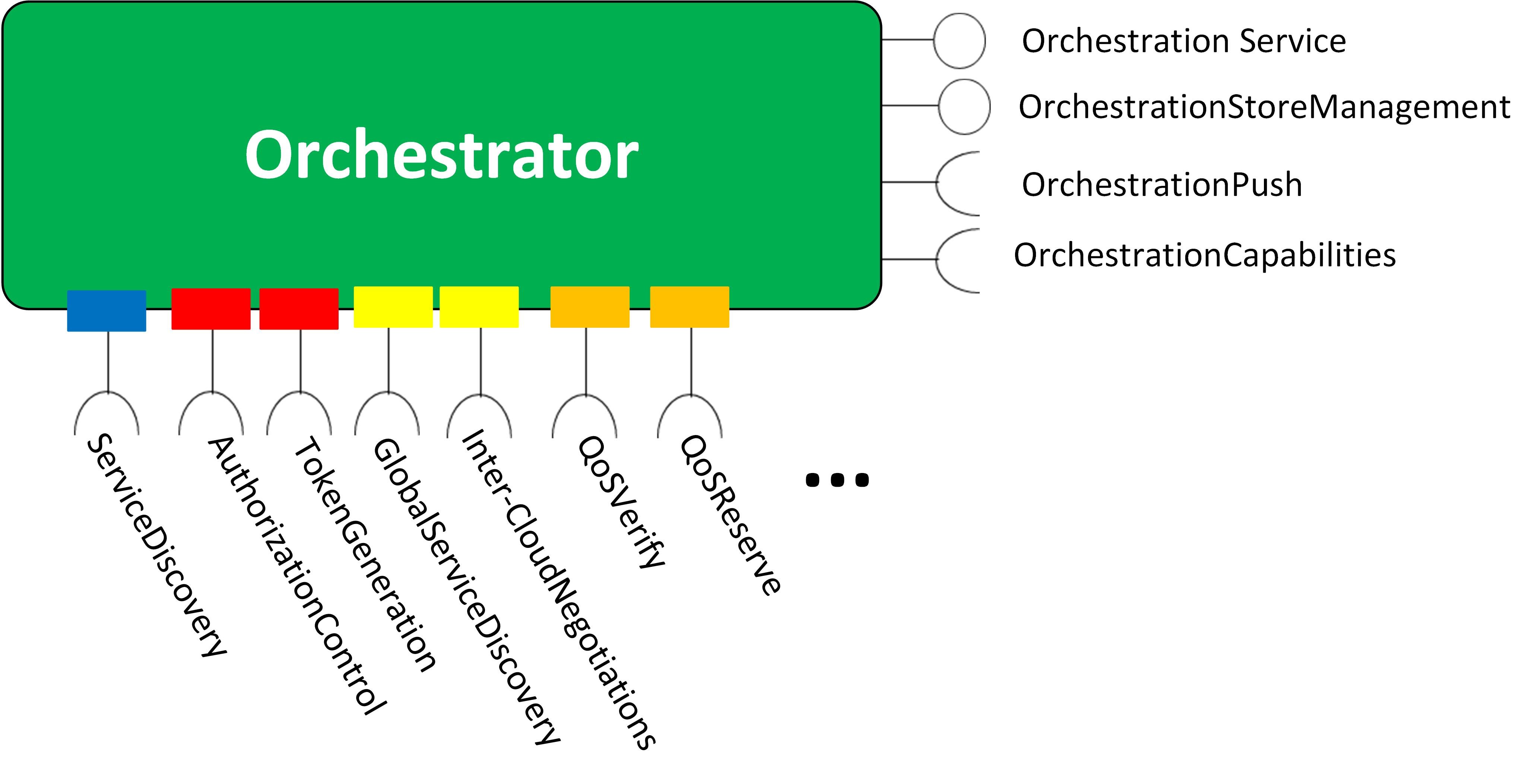


Figure Overview of the Orchestrator System

In here, the provided Services are:

* Orchestration Service
* OrchestrationStoreManagement Service

Meanwhile the consumed Services can vary, depending on the instantiation/installation of this System. For example, the Orchestrator can utilize the services of:

* ServiceDiscovery Service from the ServiceRegistry,
* AuthorizationControl Service from the Authorization System,
* TokenGeneration Service from the Authorization System,
* GlobalServiceDiscovery from the Gatekeeper,
* Inter-CloudNegotiations from the Gatekeeper,
* QoSVerify from the QoS Manager,
* QoSReserve from the QoS Manager,
* Logging services from other supporting Systems, e.g. Historian,
* and any other service from Core Systems that are necessary to settle during orchestration.

The Orchestrator mainly consumes services from other Core Systems in order to fulfill its primary functionality: provide connection targets for Application Systems in a secure and resource managed manner -- hence build an SoS.

During this *orchestration process* the Orchestrator either facilitates a service request from an Application System or processes a system-of-systems (SoS) level choreography push from the Plant Description Engine ("Choreographer"). For the latter case, the Orchestrator System consumes the OrchestrationPush from affected Application Systems in order to deliver a renewed set of connection rules to them.

Within the Orchestrator, there is a database which captures design time bindings between Application Systems, the *Orchestration Store*. Operators of the Cloud and other System-of-Systems designer tools ("SoS Choreographers") are allowed to modify the rules stored in the Orchestration Store, other generic Application Systems are not.

The ServiceDiscovery Service is used to publish the Orchestration Service in the Service Registry. This Service is also used to query the Service Registry and fetch (metadata) information on other Application Systems.

The Services of the Authorization System can be used to verify access control and implement other security-related administration tasks.

The Services of the Gatekeeper can be utilized when inter-Cloud collaboration, servicing is required.

The Services of the QoS management System can be used to manage device, network and service-level Quality of Service agreements and configurations.

1. Services and Use-cases

For the Orchestrator System, the primary scenario is to provide Application Systems with orchestration information upon request ("*service request*"). The outcome (“*orchestration response*") include orchestration rules that will tell the Application System what service provider(s) it should connect to and how.

An alternative, secondary version of this scenario involves the same information, however, provided by a connection initialized by the Orchestrator, rather than the Application Service itself ("orchestration push"). This is used to relay changes made in the Orchestration Store to the Application Systems ("changes information exchange setup within the SoS").

Another scenario is when the Orchestration Store (that stores design time orchestration-related information) of the Orchestrator is being configured via an HMI or via the Plant Description Engine (SoS Choreographer) by the operators of the Local Cloud.

Table 1: Use case 1

|  |
| --- |
| Service Request From Application System |
| ID: | ORCH-PULL |
| Brief description: | An Application System requests a Service. |
| Primary actors: | Service Consumer System |
| Secondary actors: | - the other Core System instances of the Local Cloud - the Core Systems instance of another Local Cloud (in case of inter-Cloud orchestration) |
| Preconditions: | - |
| Main flow: | 1- The Application System requests orchestration. 2- The Orchestrator System begins the orchestration process with the other Core Systems. 3- The Orchestrator System responds to the Application System based on the request. |
| Postconditions: | - |

Table 2: Use case 2

|  |
| --- |
| Orchestration information pushed to Application System |
| ID: | ORCH-PUSH |
| Brief description: | The Orchestrator pushes new information on Application Systems. |
| Primary actors: | Orchestrator |
| Secondary actors: | the other Core Systems instances of the Local Cloud |
| Preconditions: | There was a change made in the Orchestration Store. |
| Main flow: | 1- The Orchestrator detects a change in the Orchestration Store., 2- The Orchestrator begins the orchestration process with the other Core Systems for every change in the Store. 3- The orchestrator pushes new connection rules to the Application Systems based on the new Store entry. |
| Postconditions: | - |

Table 3: Use case 3

|  |
| --- |
| Orchestration information pushed to Application System |
| ID: | ORCH-PUSH |
| Brief description: | The Orchestrator pushes new information on Application Systems. |
| Primary actors: | Orchestrator |
| Secondary actors: | the other Core Systems instances of the Local Cloud |
| Preconditions: | There was a change made in the Orchestration Store. |
| Main flow: | 1- The Orchestrator detects a change in the Orchestration Store., 2- The Orchestrator begins the orchestration process with the other Core Systems for every change in the Store. 3- The orchestrator pushes new connection rules to the Application Systems based on the new Store entry. |
| Postconditions: |  |

1. Internal structure

This module is a Java jar executable. It uses the config folder contents, where the configuration files are. The code includes the following classes:

* **OrchestratorMain**: instantiates the HTTP or HTTPS server based on the properties files and command line arguments, registers into the SR
* **OrchestratorResource**: contains the REST interface functions used by Application Systems to consume the Orchestration Service
* **OrchestratorService & OrchestratorDriver**: contains the “business logic” behind the REST interface
* **StoreResource**: contains the REST interface functions used by Application Systems to consume the OrchestrationStoreManagement Service
* **CommonApi & StoreApi**: contains the REST interface functions for manipulating the underlying database tables, these interfaces are reserved for system operators (Local Cloud admins).
* **AccessControlFilter**: implements the CN-based filtering on who can access the Orchestrator Services.

# Usage

Start the module as a Java executable. The following command line arguments are available:

* “-tls”: starts the Core System in secure (HTTPS) mode, using the certificates which were set in the app.properties file.
* “-daemon” (Linux only!): starts the module in daemon mode, kill signals will prompt a normal shutdown, and the core system will de-register its services from the Service Registry.
* “-d”: starts the module in debug mode, which means every incoming REST request (URL + payload) and the corresponding response will be printed to the console output.
* “-nogk”: disables the option for Inter-Cloud orchestration, and the Orchestrator will not query the Service Registry for the Gatekeeper Services at startup (hence it does not need to be started before the Orchestrator).

1. Revision history

# Amendments

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Date | Version | Subject of Amendments | Author |
| 1 | 2017-09-29 | M2 | Initial | Csaba Hegedűs |
| 2 | 2018-02-14 | M3 | Updating document to M3 | Csaba Hegedűs |
| 3 | 2018-05-22 | G4.0 | Updated document to G4.0 | Zoltán Umlauf |

# Quality Assurance

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