Core Systems Architecture

System-of-Systems Design Document

**Abstract**

This document describes the fundamentals and the main features of the Arrowhead Framework Generation 3.2, Milestone 2.

It includes the general common elements and concepts used in the Core Systems; their technical details and gives an overview on the architecture itself.

Table of contents

[2.1. Arrowhead Services 6](#_Toc459645641)

[2.2. Arrowhead Systems 6](#_Toc459645642)

[2.3. Arrowhead Clouds 6](#_Toc459645643)

[2.4. Certificate hierarchy 8](#_Toc459645644)

[3.1. Tasks, responsibilities and scope of functionalities 12](#_Toc459645645)

[3.1.1. Service Registry (DNS-SD and REST bridge) [1] 12](#_Toc459645646)

[3.1.2. Authorization System [4] 13](#_Toc459645647)

[3.1.3. Gatekeeper System [9] 14](#_Toc459645648)

[3.1.4. Orchestration Store System [10] 15](#_Toc459645649)

[3.1.5. Advanced Orchestrator System [10] 16](#_Toc459645650)

[7.1. Amendments 23](#_Toc459645651)

[7.2. Quality Assurance 23](#_Toc459645652)

.

1. Introduction

In its evolution, Generation 1 of the Arrowhead Framework provided the fundamental principles and preliminary implementation for the mandatory core services -- whereas Generation 2 provided further core services and their mature implementations. The Generation 3.1 of the framework introduced the handling of events, security certificates, and Quality of Service constraints. Furthermore, it clarified the core services used by the application demonstrations for the Arrowhead project -- and their implementations got further polished.

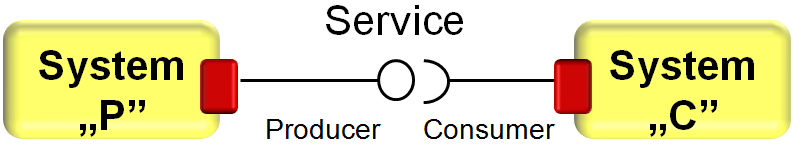
The main objective of this proof-of-concept, development-targeted generation (3.2) is to integrate fundamental solutions to the newly arisen, but essential requirements:

* advanced authentication and authorization capabilities (authorization token),
* advanced Orchestration Store functionalities
* dynamic service orchestration process,
* real-time communication (QoS) and resource allocation functionalities,
* event and alarm handling capabilities,
* the integration of the automation support services (such as the Historian or Translation Services),
* and inter-Cloud orchestration and servicing capabilities.

As per the Arrowhead approach, a Service Oriented Architecture (SOA) is established to tackle this problem: it aims at providing interoperability by facilitating the service interactions within closed or at least separated automation environments, called local clouds.

These ***Arrowhead Local Clouds*** might fulfill various tasks and can have their own sets of appointed stakeholders (e.g. their operators or developers). These Local Clouds all have their operational boundaries, let those be functional, geographical or network-segmented. Nevertheless, they must be governed through their own instances of the ***Arrowhead Core Systems***. These are clouds in the sense that they use common resources: the Core Systems of that domain.

In such Arrowhead Local Clouds there can be an arbitrary number of ***Systems*** that can provide and consume ***Services*** from one another: they create and finish ***servicing instances*** dynamically in run-time.



**Figure 1. An Arrowhead servicing instance representation**

Here, the following abbreviations, alternatives and definitions are used:

* **Arrowhead Local Cloud, Local Cloud, Cloud**: a group of (not just computational) resource bounded together to achieve a higher-level purpose and functionality (in accordance with the System-of-Systems and automation clouds concepts)
* **Service Consumer, Consumer**: The System that requests a resource from another System (initiates the communication and requests orchestration)
* **Service Provider, Provider, Producer**: The System that will serve the request from the Consumer.
* **Service offering**: when Service Providers implement a certain Service, they announce their capabilities (in the Service Registry), hence their Service offerings.
* **Servicing instance**: when two specific System instances provide and consume a specific Service from one another, this transitional connection is called a servicing instance (basically, when a Service is realized).

In this context, Arrowhead-compatible Systems are not just service endpoints, but can be realized by a wide range of devices: from a small temperature sensor, in certain use cases, up to very complex cyber-physical systems of a production plant. However, there might be operational limits of how these Systems can be handled, i.e. certain hard-wired service interactions cannot change at all, supposing that an actuator can only access certain sensors. The Arrowhead Framework has to abide by such implementational constraints that shape this world of the industrial Internet of Things.

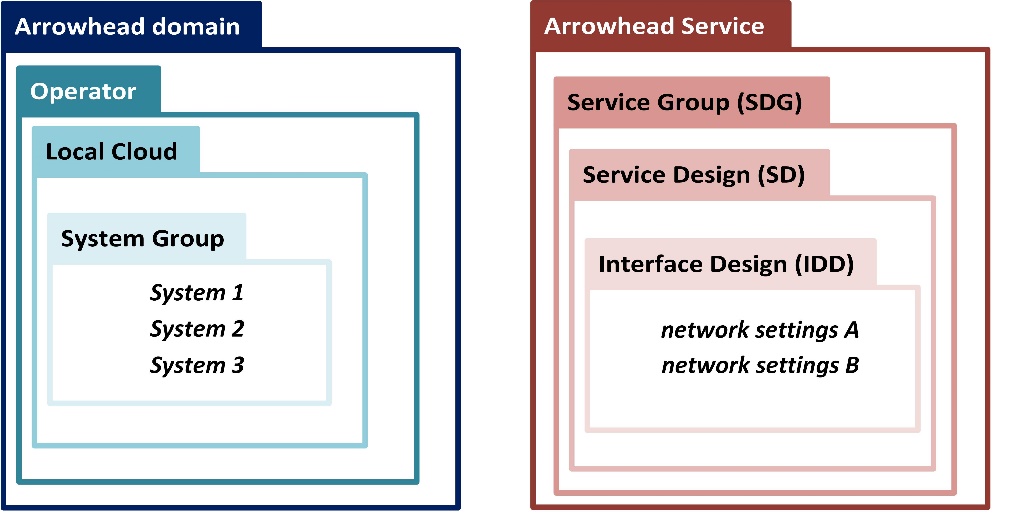
*Note: Here, the ‘device’ abstraction level is currently not used in this generation. All Arrowhead Systems are identified by IP and port. Since in cyber-physical systems (CPS), multiple network adapters, ports and resources might be present, ‘devices’ will encompass multiple Arrowhead System instances.*

**This generation’s architecture builds heavily upon the Core Systems' exclusive involvement in facilitating and managing the connections (servicing instances) within their authority: in this sense, the Core Systems are responsible for all other Application Systems and their data handled in the Local Cloud.**

**However, this feature can be disabled, and then all Core Systems are accessible separately, as previous generations, except for inter-Cloud orchestration, since that is anchored to the negotiations of the two sets of Core Systems via the Gatekeepers.**

1. Arrowhead Cloud, Systems and Services

In this generation specifically, these abstractions of cyber-physical entities and relations (local clouds, systems and services) are described within a strict hierarchy and by their own features.

****

**Figure 2. The Arrowhead System and Service hierarchies**

Throughout all communications and also within the Core Systems, these abstractions have the following descriptor objects:



**Figure 3. The Arrowhead common descriptors**

# Arrowhead Services

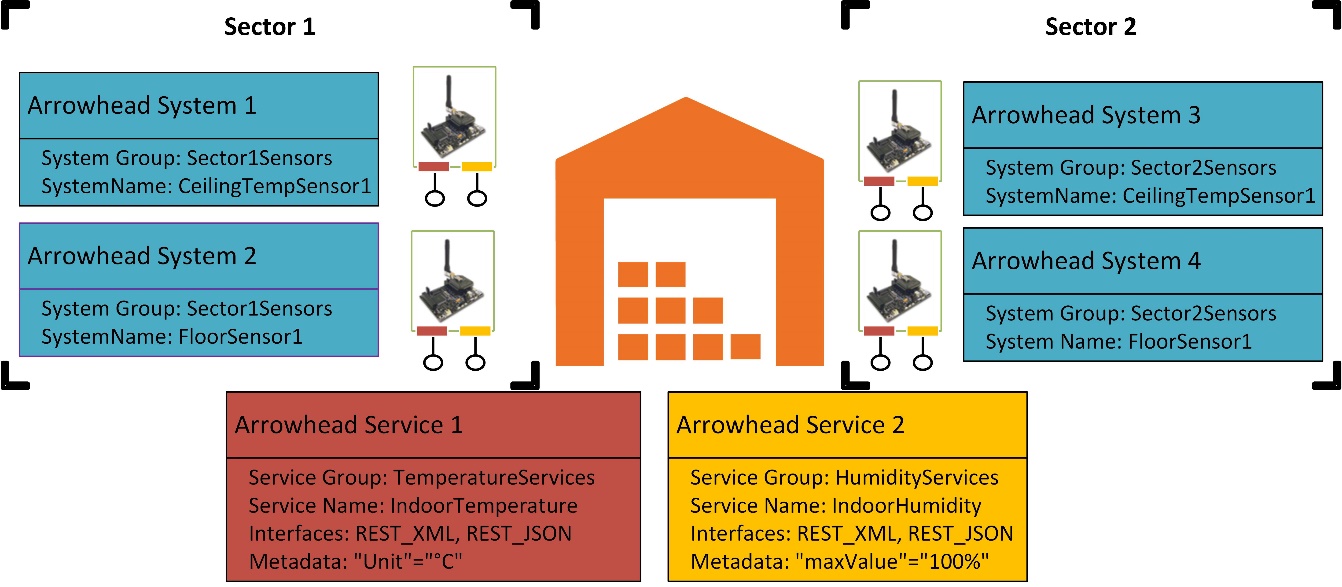
* An Arrowhead Service is identified with its name defined in its Service Description document.
* This SD defines a certain functionality (e.g. an indoor temperature readout).
* A Service belongs to a Service Group. This grouping is arbitrary and can be unique within a Local Cloud, tailored to the use case. (e.g. all temperature readout services belong to the ”TEMPERATURE” typed group of services.
* An Arrowhead Service (e.g. the indoor temp. readout) can be implemented using several interfaces (e.g. REST or CoAP) and therefore an SD can be associated with several interfaces (hence the *List<String>* for the Interfaces variable).
* An Arrowhead Service (that is e.g. requested by a System to fulfil a special task) can have Service metadata associated with it (e.g. the temperature is to be provided in °C or °F).
* This metadata is stored within each and every service offering in the Service Registry. It can be used by a customized filtering algorithm within the orchestration process to filter out unsuitable Service Providers that don’t fit the requirements.
* Currently key-value pairs can be added as metadata (e.g. ”unit”=”◦C”).
* This will enable the integration of standardized data descriptors, e.g. SenML or IPSO Smart Objects.

# Arrowhead Systems

* This descriptor identifies an Application System within its own Cloud.
* Systems are grouped to **systemGroups** (e.g. whether they belong to the same production line in a plant). This logical grouping is also free of choice. Every System has its **systemName**.
* Systems have their own IP **address** and **port**, where they offer Services (an Arrowhead System is then bound to an http server in this REST implementation listening under one port).
* Systems can provide several Services (and even multiple instances of the same Service, with the same or different interface), therefore **Service URL-s are not associated with the Systems themselves, but stored separately**.
* However, authentication information can be stored for Systems (e.g. in the Authorization System). This field (**AuthenticationInfo** String) is only used to store the X.509 certificate public key serialized in the MySQL database for the Authorization System. This field is **not mandatory** for the REST communications*.*

# Arrowhead Clouds

* These classes help the identification of other Clouds for the local Systems.
* These classes are only used by the Core Systems while they establish (inter- or intra-Cloud) service connections (servicing instances)
* An Arrowhead Cloud belongs to an **operator** (e.g. its company) and has a unique **name**.
* These names (cloudName and operator’s name) are also arbitrary.
* The sole point of external contact to a Cloud is its Gatekeeper Core System.
* Therefore we store the Gatekeeper access path of an other Cloud. The schema is <http://gatekeeperIP:port/gatekeeperServicesExternalURI/>.
* We store the authenticationInfo (*placeholder* String for now) for each neighbor Cloud, *placeholder* String for now (don’t care value). This field is not mandatory in the REST communications.



**Figure 4. The Arrowhead common descriptors in practice**

For understanding the naming convention, let us consider temperature and humidity sensors in a refrigerator warehouse. They measure the indoor temperature at different locations and heights. Each sector has two sensors that measure the floor- and ceiling-level temperatures. Figure 4 shows how these sensors are represented as Arrowhead Systems and how their measurement readout resources can be abstracted as Arrowhead Services.

These common descriptors appear in all the core service REST calls, in the payloads formatted in JSON.

An example ArrowheadSystem description in JSON:

**{**

"systemGroup"**:** "Sector1Sensors"**,**

"systemName"**:** "CeilingtempSensor1"**,**

"address"**:** "192.168.1.111"**,**

"port"**:** "8080"

**}**

An example ArrowheadService (its fully detailed form[[1]](#footnote-2)):

**{**

"serviceGroup"**:**"TemperatureServices"**,**

"serviceDefinition"**:**"IndoorTemperature"**,**

"interfaces"**:[**

"RESTXML"**,** "RESTJSON"

**],**

"metaData"**:[**

**{**

"key"**:**"unit"**,**"value"**:**"C"

**},**

**{**

"key"**:**"maxValue"**,**"value"**:**"100"

**}**

**],**

**}**

An example Arrowhead Cloud representation:

**{**

"operator"**:** "AITIA"**,**

"cloudName"**:** "TestCloud1"**,**

"address"**:** "99.99.99.99"**,**

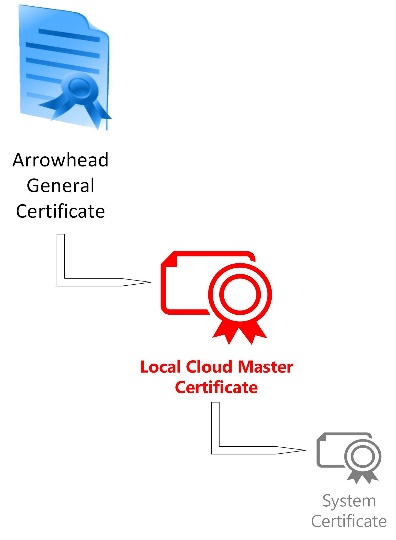
"port"**:** "8080"**,**

"gatekeeperServiceURI"**:** "/gatekeeper"

**}**

# Certificate hierarchy

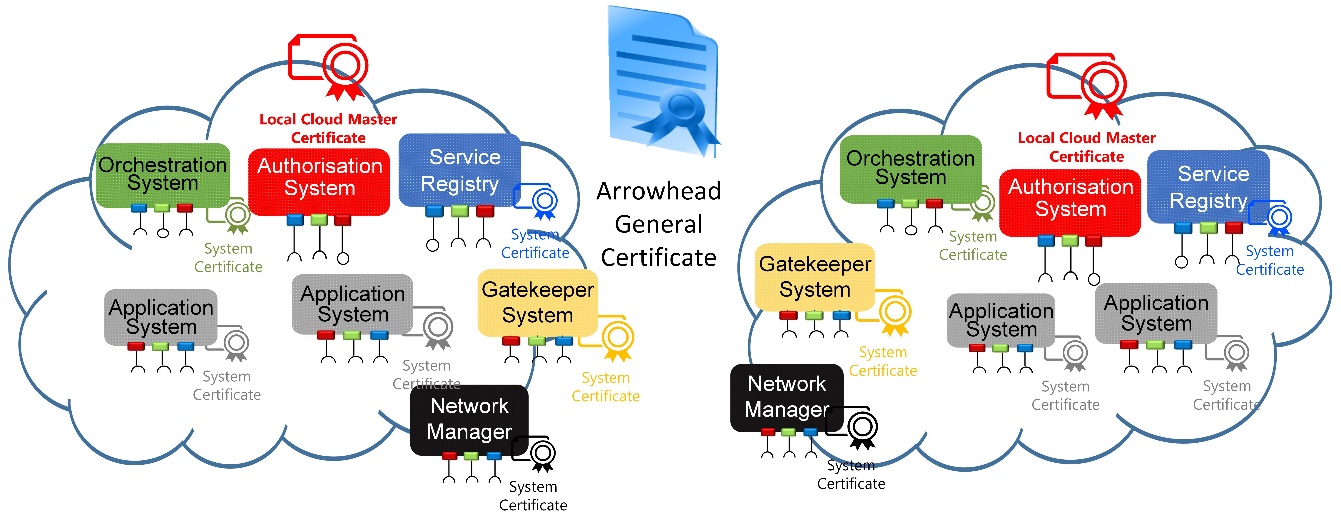
This implementation uses X.509 certificates to establish secure (SSL) connections between entities (resources), if enabled. Every Arrowhead System (and the core ones, as well) should have its own certificate, specifically created from a master Arrowhead certificate.

This certificate chain represents the unified Cloud and System architecture and enables advanced authorization capabilities (such as identitiy verification and access control with authorization tokens).

From a master Arrowhead certificate (currently self-signed), every Local Cloud can spawn their own certificate (the master certificate signs the Cloud one).

This process is to be governed by the Arrowhead consortium when new Local Clouds are established. Clouds can only participate in inter-cloud interactions when their cloud certificate is signed by the master Arrowhead CA (only manual generation and signature is used).

Within the newly established Local Clouds, Systems must also have their own certificate, that is signed by their own Cloud certificate. This signature ensures that a System ”belongs” to the Cloud and is properly initiated (bootstrapped into the Cloud).



**Figure 5. The Arrowhead common descriptors in practice**

Identity verification and admission control utilizes the Common Name (CN) field of the subject name in a X.509 certificate. The current (test) certificates use RSA 2048 bits and are stored in Java KeyStore files separately.

An Arrowhead-compliant CN structure represents the System hierarchy, separated by dots (”.”):

CN= ***<systemName>.<systemGroup>.<cloudName>.<operatorName>.arrowhead.eu***

For example:

CN= *Orchestrator.CoreSystems.TestCloud1.aitia.arrowhead.eu*

OU=*SGA* O=*AITIA INC*.

L= *Budapest* ST=*HU*

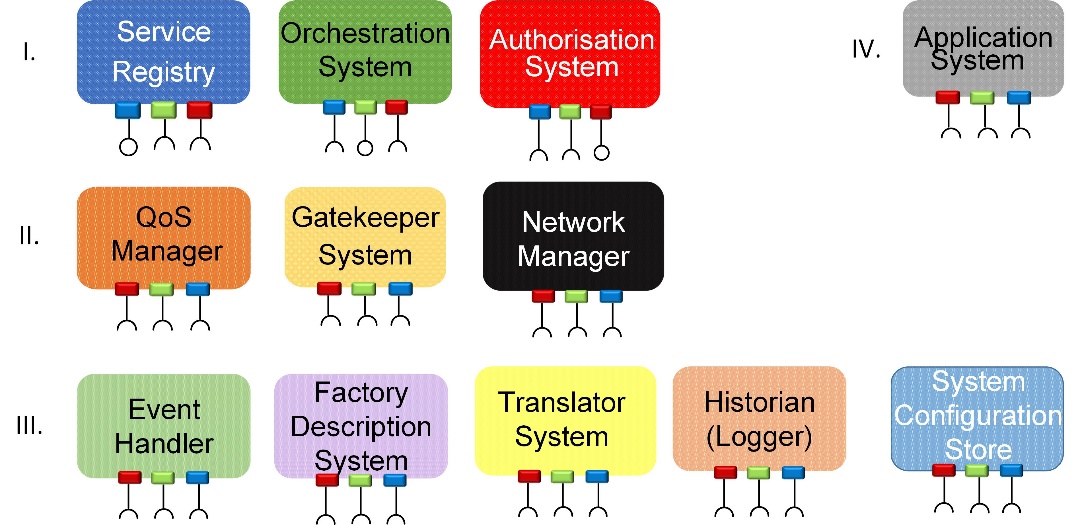
C=*HU* E= [*hegeduscs@aitia.ai*](mailto:hegeduscs@aitia.ai)

**Beware**: The Arrowhead Core Systems must belong to the” CoreSystems” system group and must share the same Cloud information. The CN field is handled regardless of upper casing.

1. Arrowhead Core Systems

As per the Arrowhead approach, Local Clouds are governed through their own instances of the Core Systems. There are two main groups of the Core Systems:

* the mandatory ones that need to be present in each Local Cloud, and
* the automation supporting ones that further enhance the core capabilities of a Local Cloud.



**Figure 6. The Arrowhead Core Systems in scope**

This implementation encompasses the following systems:

* Service Registry (DNS-SD BIND server with a REST bridge)
* Service Registry (for development purposes: MySQL server with a REST bridge)
* Orchestration Store System
* Orchestrator System
* Authorization System
* Gatekeeper System
* Generic Application System skeletons (as Service Provider and Consumer as well)

These Systems are fully implemented and available.

These Systems consume and provide the following Arrowhead Services from one another:

***Note: Not all of these Core Services are documented as fully fledged Arrowhead Services as some of them are not available for Application Systems, since they are just part of the internal communications between the Core Systems. Fully documented Core Services are:***

* ***OrchestrationServiceM2 REST-JSON-TLS***
* ***ServiceDiscoveryM2 REST-JSON-TLS***
* ***SimpleTemperature***

***The use of other Core Services within Application Systems is not advised, but when the orchestration process is disabled, they can be implemted.***

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | **System** (black box, SySD level) | Service Registry (DNS-SD) | ServiceRegistry REST | Authorisation System | Orchestrator System | Gatekeeper System | Application systems |
| **Services** |  |  |  |  |  |  |  |
| Service Discovery DNS-SD |  | P | C |  |  |  | C |
| Service Discovery REST-JSON-TLS |  |  | P |  |  | C | C |
|  |  |  |  |  |  |  |  |
| OrchestrationM2 REST-JSON-TLS |  |  |  |  | P |  | C |
|  |  |  |  |  |  |  |  |
| AuthorizaitonControlM2 REST-JSON-TLS |  |  |  | P | C | C |  |
| TokenGenerationM2 REST-JSON-TLS |  |  |  | P | C |  |  |
|  |  |  |  |  |  |  |  |
| Global Service Discovery |  |  |  |  |  | P |  |
| Inter-Cloud Negotiations |  |  |  |  |  | P |  |
|  |  |  |  |  |  |  |  |
| SimpleTemperature Service |  |  |  |  |  |  | P/C |

**To conclude**, Application Systems only the Service Registry Service and the Orchestration Service to get orchestration information in run-time.

This approach alleviates the need to implement multiple Core Services interfaces in Application Systems: they only have to announce what Service they offer in the Service Registry and use the Orchestration Service to acquire orchestration information for operation in run-time.

***Note:*** It is fully transparent to Service Consumers whether the actual, orchestrated Service Provider is within its own Local Cloud or not. The network visibility and other addressability issues are currently resolved here (as per M2, to be . Conceptually, this data tunneling between networks is the tasked with the Gateway (in concept). It will be implemented within the M3 release*.*

# Tasks, responsibilities and scope of functionalities

## Service Registry (DNS-SD or MySQL based REST module) [1]

In the color coding of the Core Systems on figures, the Service Registry (and its Services) bear the blue color.

The Service Registry bears the following responsibilities:

* It has to store all available Service offerings from currently active Systems.
* Systems have to announce their presence in it, and the services they can offer.
* The Service Registry has to implement the first line of authorization control (using the TSIG key).

The Service Registry stores the following information:

* Systems that have active Service offerings
* What Services are offered by each Systems
* Every offered Service instance has a list of implemented interfaces and Service metadata associated with it

The Service Registry has the following capabilities:

* Systems can register and revoke service offerings
* Periodical cleanup of the database to sort out invalid (offline) offers
* Detailed logging of operation

The Service Registry offers the following public Services for Application Systems:

* ServiceDiscovery Service [2]

The Service Registry is implemented:

* DNS-SD BIND server or a MySQL stores the entries
* Grizzly container based Jersey REST resources written in Java
* DNS-SD Java library or Hibernate ORM with MySQL connector
* Periodical cleanup based on the success of a dummy HTTP/GET request to the IP:port entry (TimerTask of Java)
* Using SSL or unsecured
* Detailed logging in SQL table (uses Log4J)

The Service Registry has the following known limitations:

* It should rely on the DNS-SD server for release versions
* The development-aiding version relies on MySQl to support quick start with the Framework

## Authorization System [4]

In the color coding, the Authorization System (and its Services) is red.

As its name suggests, this Core System handled AA functionalities. It has two major scope of functionalities:

The Authorization System bears the following responsibilities:

* It provides the Orchestrator and Gatekeeper with authorization status of subjects (AuthorizationControlM2 service)
* It generates authorization tokens during the orchestration process

The Authorization System offers the following public Services for Application Systems by default: **none.**

The following Services are provided primarily for the Orchestrator System:

* AuthorizationControlM2 Service [5]
* TokenGeneration Service [7]

The Authorization System stores the following information:

* It stores authentication information for Application Systems in the Local Cloud (their certificate public key serialized in Base64 encoding)
* It stores intra-cloud access rights information for all Application Systems in the Local Cloud (which Systems as Consumers can access what Services at which Service Providers)
* It stores inter-cloud access rights information (which Local Clouds are authorized to consumer Services from this Cloud’s Systems)

The Authorization System has the following capabilities:

* To tell whether a given Consumer can access a Service of a Producer
* Generate an authorization token that validates this right and verifies that proper orchestration took place

The Authorization System is implemented:

* Grizzly container based Jersey REST resources written in Java: it has its own port and URL structure
* MySQL server with Hibernate ORM
* Using SSL or unsecured; implemented with X.509 certificates and RSA encryption
* Detailed logging in SQL table (uses Log4J)

The Authorization System has the following known limitations:

* The master Arrowhead certificate is currently self-signed (should be signed by a trusted global CA, such as VeriSign)
* It has a static database that is shared with the other core systems

## Gatekeeper System [9]

In the color coding, the Gatekeeper (and its Services) bear the yellow color.

This System handles inter-Cloud service interactions by providing two core services to the Orchestrator.

The Gatekeeper System bears the following responsibilities:

* Extends the Service discovery capabilities outside the Local Cloud
* Helps to establish Service connections by aiding the inter-cloud orchestration process
* Handles preferences in its two, above mentioned processes

It offers the following public Services for Application Systems: **none.**

It offers the following Services for the Orchestrator:

* GlobalServiceDiscovery
* Inter-CloudNegotiations

The Gatekeeper System stores the following information:

* The neighborhood: which other Local Clouds are connected and can request Services from

The Gatekeeper System is implemented:

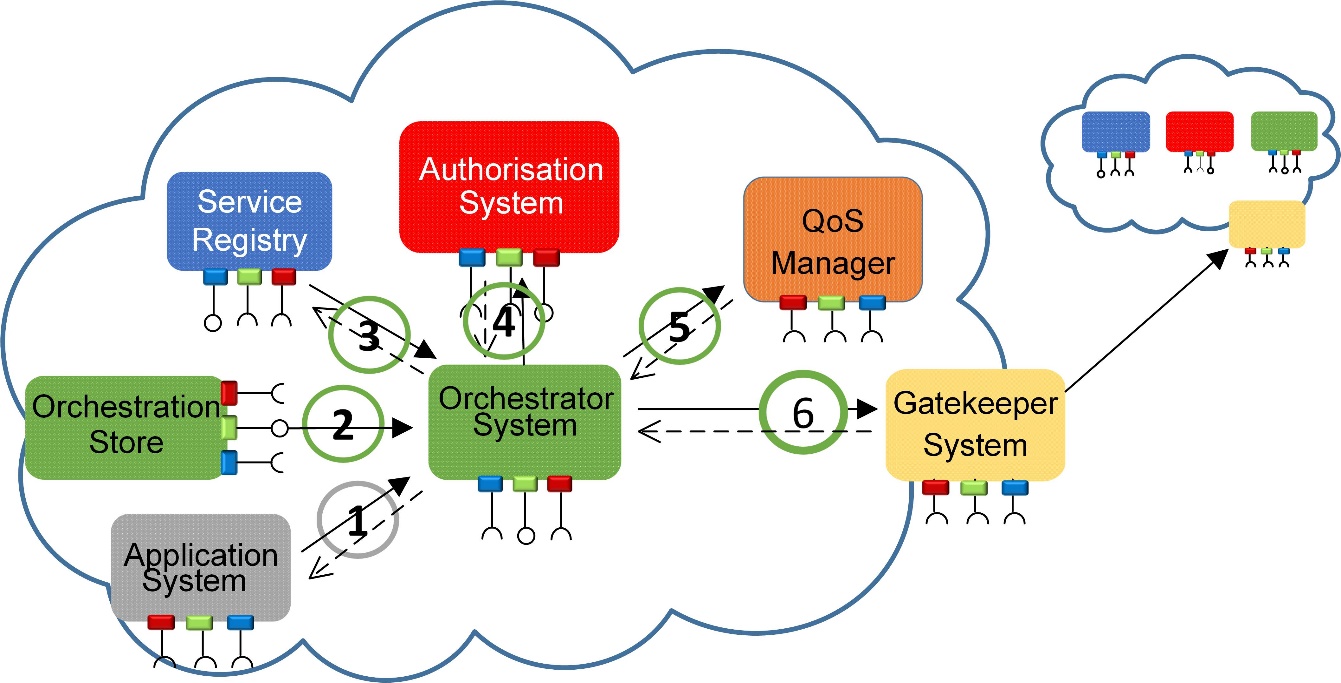
* Grizzly container based Jersey REST resources written in Java with Hibernate ORM
* Using SSL or unsecured (communications between Gatekeepers are currently not secured)
* Detailed logging in SQL table (uses Log4J)

The Gatekeeper System has the following known limitations:

* Currently lack proper separation of its resources that are offered outwards and inwards to the Clous

## Orchestrator System [10]

This Core System is used to fetch a special type of configurational data during run-time. In this generation, the Orchestrator bears the biggest responsibility in a Local Cloud. It is the primary decision-maker that is aware of the current conditions in the SoS. Its primary task is to allocate Service Providers to the Service Requests sent in by Systems. During this ***orchestration process*** the Orchestrator consults with the other Core Systems and makes a decision based on the responses.



**Figure 7. Overview of the Orchestration process**

This process is initiated by Application Systems, when one submits a Service Request Form which describes what is requested and how. At the end, they receive an Orchestration Response, which details where the Consumer needs to connect to and other parameters necessary. Further details can be found in [10] and [11].

*Note: As mentioned above, this orchestration is requested by Application Systems (“client pull-based”). Central governance can be built upon this architecture, where Application Systems receive periodic event updates (using the EventHandler System) that triggers re-orchestration requests in Application Systems. The EventHandler Core System is not part of the G3.2 M2 codebase directly.*

Here, Application Systems have an increasing degree of freedom on requesting orchestration-typed configuration based on their own capabilities. This version of the Arrowhead framework defines and targets the following layers of Application System behavior based on their autonomy:

* **Passive Service Providers**: Such Application Systems only offer certain services. They only have to register these in the Service Registry and respond to inbound servicing requests. For example, “dumb” sensors, that can provide readouts.
* **Statically pre-configured Service Consumers**: In certain cases, an Application System can suffice its purpose by periodically consuming the same set of Services from hardwired set of Service Providers. This can happen e.g. in data logging devices that read out and store measurements.
* **Service Consumers with statically configured Service Providers**: they consumer Services automously in run-time (according to their implemented operational logic), but they can only consume certain Services from certain hardwired lists of Service Providers. This is the case e.g. for processing units that can only access certain sensors or their backups (for example actuating can only happen based on local temperature sensors – but if one fails, there has to be an auxiliary one).
* **Fully autonomous Service Consumers**: they only require advanced matchmaking if they are seeking Service Providers. This matchmaker orchestration process only has to return with an appropriate (suitable) Provider and make resource reservations and run other centralized tasks if necessary. This is required in a market-like environment, where (e.g. energy) offers have to meet demands.
* **Self-orchestrating Systems**: they implement all core service interfaces on their own and capable of self-orchestration using the appropriate processes. Centralized resource management might still be required.

From the technical point of view, the Arrowhead-compliance maturity model is considered valid.



**Figure 8. Arrowhead Maturity levels**

1. Concepts not covered by M2

This section covers questions regarding missing functionalities, components and their possible solutions – some only on a conceptual level. The current set of Core Systems and Services naturally don’t satisfy all requirements that can turn up in use cases – but this list shows possible guidelines.

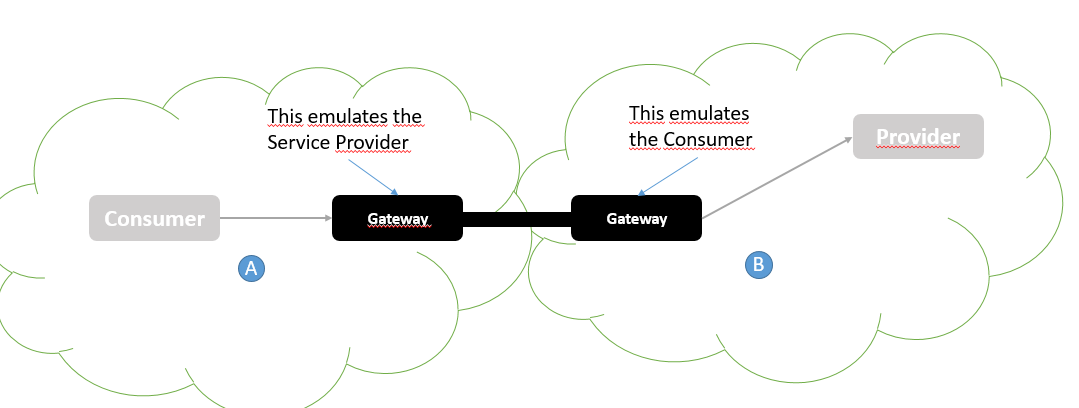
1. **Network visibility problems**

Local Clouds are defined by having their own boundaries. These might include having a closed communication network, where IP addresses are local and firewalls (or even full physical segregation) separates Systems from the outside world. Within these Local Clouds network visibility and addressability problems are not considered.

However, inter-Cloud communications require that Application Systems can address, send to and receive data from external entities. It is also required that these interactions happen in a controlled way and should not compromise the Local Cloud’s integrity, security or safety.

These issues are mended within the inter-Cloud orchestration process. In this architecture, the Core Systems are responsible for Application Systems: inter-Cloud servicing cannot happen without the two set of Core Systems knowledge and approval. Since Local Clouds are generally autonomous and operationally independent from each other – and might not even share stakeholders, proper orchestration is required for establishing every servicing instances.

In this process, the last step is building up a data path between the to-be Consumer and Provider that reside in two different Clouds. This data path requires gateways on the edges of both Local Clouds. Such gateways replicate the servicing partner at the borders and tunnel the communication in-between Clouds.



**Figure 8. Tunneling the communication in inter-Cloud scenarios to maintain closeness of Local Clouds**

***Note: Currently, a such Gateway System (previously called Network Manager), is not implemented as per M2. It is under development, will be released within Milestone 3 of G3.2.***

1. **Resource Management, Quality-of-Service**

Managing the available (not just computational) resources within a Local Cloud is yet to be solved. Resources in a Local Cloud can vary from e.g. processor task time slots of embedded Systems through real-time networking QoS up to setting client-server servicing requirements (e.g. transactions per second).

The advanced orchestration process showcases the integration with the QoS Manager. Its two main features are:

* the verification of the requirements set by new Service Requests at the early stage of the orchestration process, and
* the actual resource reservation process with configuring the affected components as the last step of the orchestration process.

Therefore, this optional Core System is in tight cooperation with the other Core Systems, and it also participates in the orchestration process.

However, it is worth noting that the QoS Manager concept is yet restricted to only handle communicational QoS.

For further details, see [REF].

1. **Push Orchestration,** re-orchestration, error handling – integration with the Event Handler System

Currently, the Orchestrator is request-triggered. Hence, error and event handling is not in its scope. If a servicing instance breaks up (e.g. the Provider vanishes in an instant), it is the partners’ task to signal the appropriate Core Systems (e.g. QoS Manager or the Historian) and request re-orchestration. Through this approach, the scalability of a Local Cloud is maintained, since there is no centralized monitoring of every App. System in run-time (that would require processing power and generate unnecessary periodic control traffic. If, however, needed, an additional Core System has to be defined for this task and be integrated with the mandatory ones.

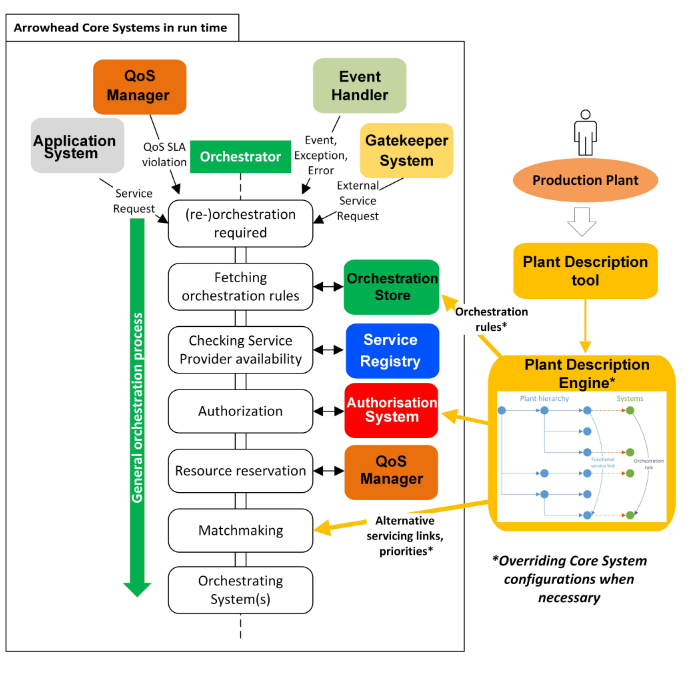
1. **System-of-System Choregraphy**

These Core Systems currently run on pre-defined rulesets (in the Orchestration and Authorization stores and in the Service Registry). They also assume that Application Systems themselves are configured to fulfil their tasks – in somewhat autonomous way.

Here, orchestration refers to a supportive task, namely providing instructions to Application Systems on their service consumption behaviour. Since this is merely a black-box definition for the Orchestration service (describing its Service: inputs, outputs and interfaces), it provides room for various technological approaches on its internal logic and implementation.

Therefore, a high-level design decision can be made on how to build the orchestration process that best matches service-consuming Systems to producing Systems. This degree of freedom can result in fundamentally different “Orchestrators” that fit the different needs of various use cases. Currently, there are three archetypes of Orchestrators in scope within Arrowhead, as discussed in [10].

Building on this, an additional engine can be introduced that translates process engineering targets into System configuration – both for Application and as well for Core Systems.



**Figure 9. A System-of-Systems configuration engine -a proposal**

***Note:*** *This concept is under design. Will be release in a later time.*

1. Deployment: bootstrapping, authentication and authorization into a Local Cloud

This issue is not covered by this generation of the Framework. Security is established with a manual generation and integration of X.509 certificates into Application Systems and authorization is handled via issuing signed and encrypted tokens to validate Consumers.

High level concepts regarding the bootstrapping of newly awakened Systems and run-time dynamic initialization is discussed in the Arrowhead book.

***Note: The SystemRegistry and DeviceRegistry systems will be used for this process.***

1. **Integration of Automation Supporting Core Services (translation)**

Protocol translation is currently not integrated within this generation of the framework.



**Figure 10. A System-of-Systems configuration engine -proposal**

1. **Metadata handling**

As indicated in the description of the Arrowhead Service object, metadata can currently be associated with every instances of the Service separately.

Metadata is stored in the Service Registry and enables smart filtering on Service Providers.

Therefore it is worth noting, that only Service instance related metadata is supported within G3.2 M2.

1. **Arrowhead System definitions issue**

As indicated before, the current Arrowhead System definition does not cover all aspects of CPS-s. The ‘device’ level of abstraction might be necessary to introduce or the System definition will have to be augmented.

The usage of the devices level definition is not implemented in G3.2 M2.

1. Non-functional Requirements

This architecture currently does not have any further requirements.

1. References

[1] ServiceRegistry System SysDD

[2] ServiceDiscoveryM2 SD

[3] ServiceDiscoveryM2 Service IDD – REST-JSON-TLS

[4] Authorization System SysDD

[7] Authorization Token Service SD

[8] Authorization Token Service IDD - REST-JSON-TLS

[10] Orchestrator System SysDD

[11] OrchestrationM2 Service SD

[12] OrchestrationM2 Service IDD REST-JSON-TLS

1. Revision history

# Amendments

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Date | Version | Subject of Amendments | Author |
| 1 | 2016-08-19 | 0.1 | Release ready | Csaba Hegedűs |
| 2 | 2017-10-02 | M2 | Updated for M2 | Csaba Hegedűs |

# Quality Assurance

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

1. Arrowhead Services are identified by their ServiceGroup-ServiceName fields by default. However, the full description requires other fields in practive, e.g. when a System offers an implementation out in the Service Registry. [↑](#footnote-ref-2)