Eclipse Arrowhead Architecture

System-of-Systems Design Document

**Abstract**

This document describes the fundamentals and the main features of the Arrowhead Framework v 4.1.3.

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

1. Introduction

Requirements

Essential requirements:

* Seamless interoperability between IoT systems
* Real time performance
* Security
* Interoperability
* Scalability
* SoS integration
* Flexible SoS
* Evolvable SoS
* Engineering simplicity, process and tool chains
* Enable realisation of complex automation architectures (RAMI4.0, IIRA)

Fundamental structure

* Local clouds [Delsing 2016][Delsing 2017]
  + Self contained
  + Protective shell around it’s functionality
  + Read time and security
  + Inter cloud service exchanges

***The Arrowhead Local Cloud Concept***

A Local Cloud have a fully private network. The local cloud is self contained, thus having all the necessary technology to fulfil its tasks. If the private network has real time capabilities hard real time control loops can be implemented.

The private network Meaning that no external network traffic can enter the local cloud without approval.

* SOA [IBM kille]
  + Look up
  + Late biding
  + Loosely coupled
  + Protocol agnostic
  + Encoding agnostic
  + Semantic agnostic
  + Compression agnostic
* Architecture terminology
  + [Delsing 2017]
* Documentation system
  + [Blomqvist 2015]
* Security [Varga 2017][Delsing 2017]
  + Authentication
  + Authorisation
  + Audit
  + Certificates based on X.509
  + Certificate distribution
  + Onboarding procedure
* Naming convention
  + [Paniaqua 2019]
* Interoperability
  + Translation
    - Protocols
    - Encodings
    - Compression
    - Semantics

**Not in SoSD part of the book**

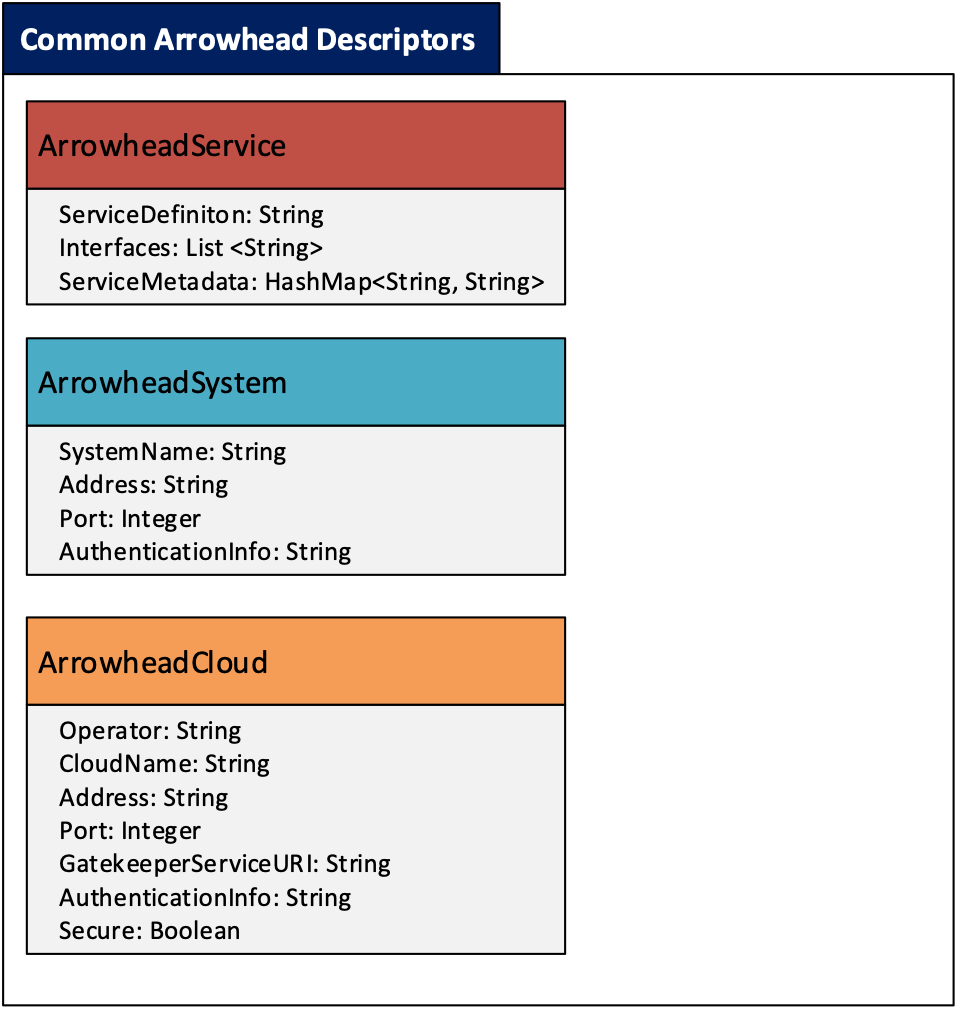
* Engineering model
  + SysML
  + Engineering process
  + Engineering tools
  + Tools chains

Core systems

* Core system relation to requirements
* Engineering process
* Tools
* Tool chains

1. Arrowhead Cloud, Systems and Services

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



**Figure 2. The Arrowhead common descriptors**

* 1. **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).
* 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”=”celsius”).
* This will enable the integration of standardized data descriptors, e.g. SenML or IPSO Smart Objects.
  1. **Arrowhead Systems**
* This descriptor identifies an Application System within its own Cloud. 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 System objects 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*.*
  1. **Arrowhead Local Clouds**
* This descriptor helps the identification of other Clouds for the local Systems.
* 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 another Cloud. The schema is http://gatekeeperIP:port/gatekeeperServicesExternalURI/.
* We store the authenticationInfo (*placeholder* String for now) for each neighbor Cloud. This field is not mandatory in the REST communications.

An example ArrowheadSystem description in JSON:

{

"systemName": "InsecureTemperatureSensor",

"address": "127.0.0.1",

"port": 8454

}

An example ArrowheadService description in JSON

{

"serviceDefinition": "IndoorTemperature",

"interfaces": ["json"],

"serviceMetadata": {

"unit": "celsius"

}

}

An example ArrowheadCloud description in JSON:

{

"operator": "SmartGridOperator",

"cloudName": "SmartGrid",

"address": "10.0.0.82",

"port": 8446,

"gatekeeperServiceURI": "gatekeeper",

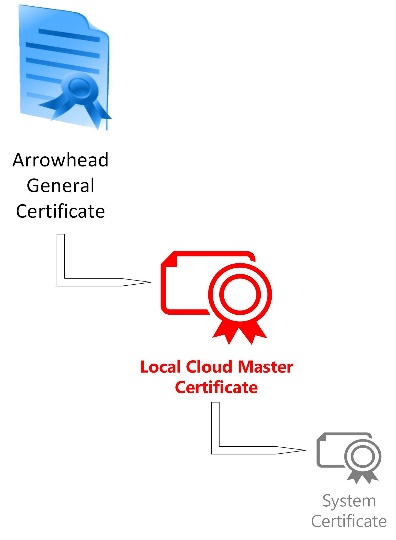
"authenticationInfo": "Gatekeeper PK",

"secure": false

}

* 1. **Certificate hierarchy**

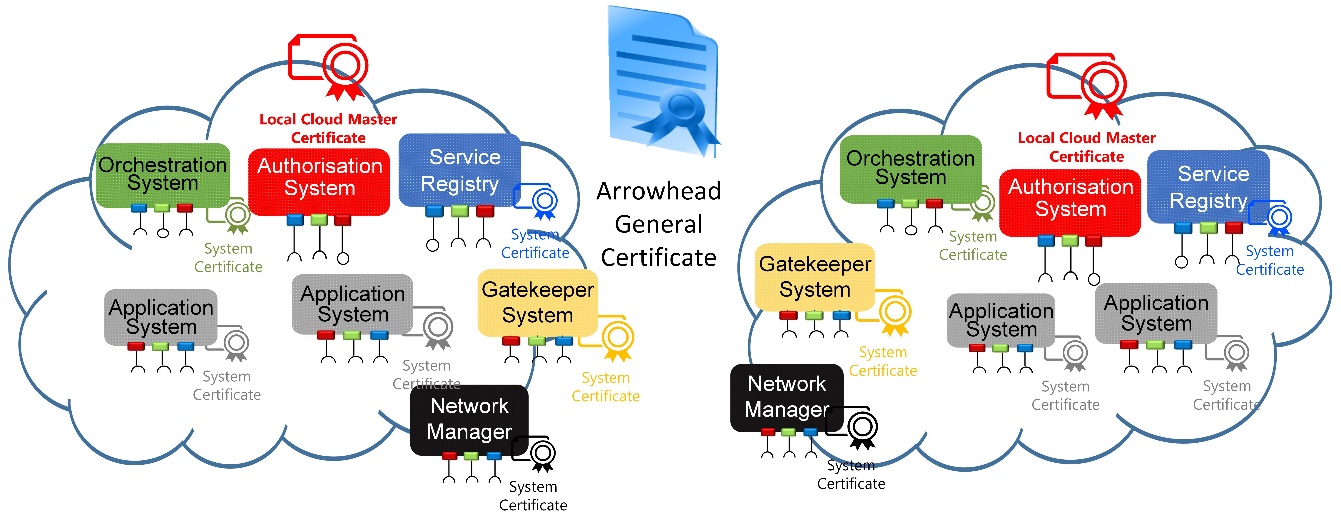
This implementation uses X.509 certificates to establish secure (TLS) connections between entities (resources), if enabled. Every Arrowhead System (and the core ones, as well) should have its own certificate, specifically created from a master Arrowhead certificate.

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 3. The Arrowhead certificate structure 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 is structured the following way, separated by dots (”.”):

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

For example:

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

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

L= *Budapest* ST=*HU*

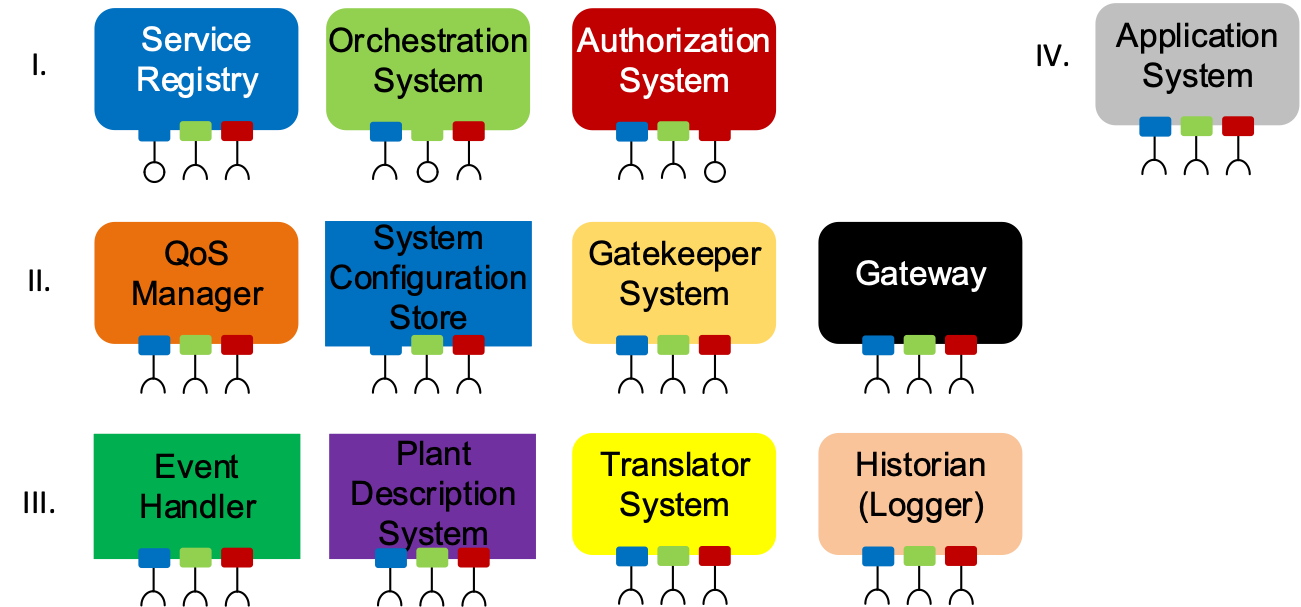
C=*HU* E= *[hegeduscs@aitia.ai](mailto:hegeduscs@aitia.ai)*

The CN field is handled regardless of upper casing (not case sensitive).

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.



**F****i****g****ure 4. The Arrowhead Core Systems in scope**

This implementation encompasses the following systems:

* Service Registry (DNS-SD BIND server with a REST bridge)
* Service Registry (MySQL server with a REST bridge)
* Orchestration Store System
* Orchestrator System
* Authorization System
* Gatekeeper System
* Gateway System
* Event Handler 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:***

* ***OrchestrationService***
* ***ServiceDiscovery***

***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)** | **Service Registry REST** | **Authorization System** | **Orchestrator System** | **Gatekeeper System** | **Application systems** |
| **Services** |  |  |  |  |  |  |  |
| **Service Discovery DNS-SD** |  | P | C |  |  |  | C |
| **Service Discovery REST-JSON-TLS** |  |  | P |  |  | C | C |
|  |  |  |  |  |  |  |  |
| **Orchestration Service** |  |  |  |  | P |  | C |
|  |  |  |  |  |  |  |  |
| **Authorization Control** |  |  |  | P | C | C |  |
| **Token Generation** |  |  |  | P | C |  |  |
|  |  |  |  |  |  |  |  |
| **Global Service Discovery** |  |  |  |  | C | P |  |
| **Inter-Cloud Negotiations** |  |  |  |  | C | P |  |
|  |  |  |  |  |  |  |  |
| **Session Establish** |  |  |  |  |  | C |  |
| **Session Management** |  |  |  |  |  | C |  |
|  |  |  |  |  |  |  |  |
| **Event Publishing** |  |  |  |  |  |  | C |
| **Event Subscription** |  |  |  |  |  |  | C |
|  |  |  |  |  |  |  |  |
| **Simple Temperature Service** |  |  |  |  |  |  | P/C |

To conclude, Application Systems only use the Service Registry Service and the Orchestration Service to get orchestration information in run-time. (Event Handler Services are not part of the orchestration process).

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 resolved through the Gateway services (black).

* 1. **Tasks, responsibilities and scope of functionalities**
     1. **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 must 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
  + 1. **Authorization System [4]**

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

As its name suggests, this Core System handles 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 (AuthorizationControlM3 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:

* AuthorizationControl 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)
  + 1. **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 TLS or unsecured
* Detailed logging in SQL table or file (uses Log4J)

The Gatekeeper System has the following known limitations:

* n.a.
  + 1. **Gateway System [13]**

In the color coding the Gateway (and its Services) are marked with the black color. This System is responsible for creating a data path between Application Systems in different Local Clouds, where network visibility is a problem. This is done with the help of a AMQP message broker.

The SessionEstablish service is used by the Gatekeepers (on both consumer and provider sides). During an inter-Cloud servicing instances, application systems (although they are not aware of this) are communicating with the Gateway System, through a socket dedicated for the given servicing instance. The Gateway Systems communicate through an AMQP message broker, forwarding the requests and responses of the application systems.

In secure mode, the Gateway Systems use payload encryption on the messages, in order to defend against malicious AMQP message brokers.

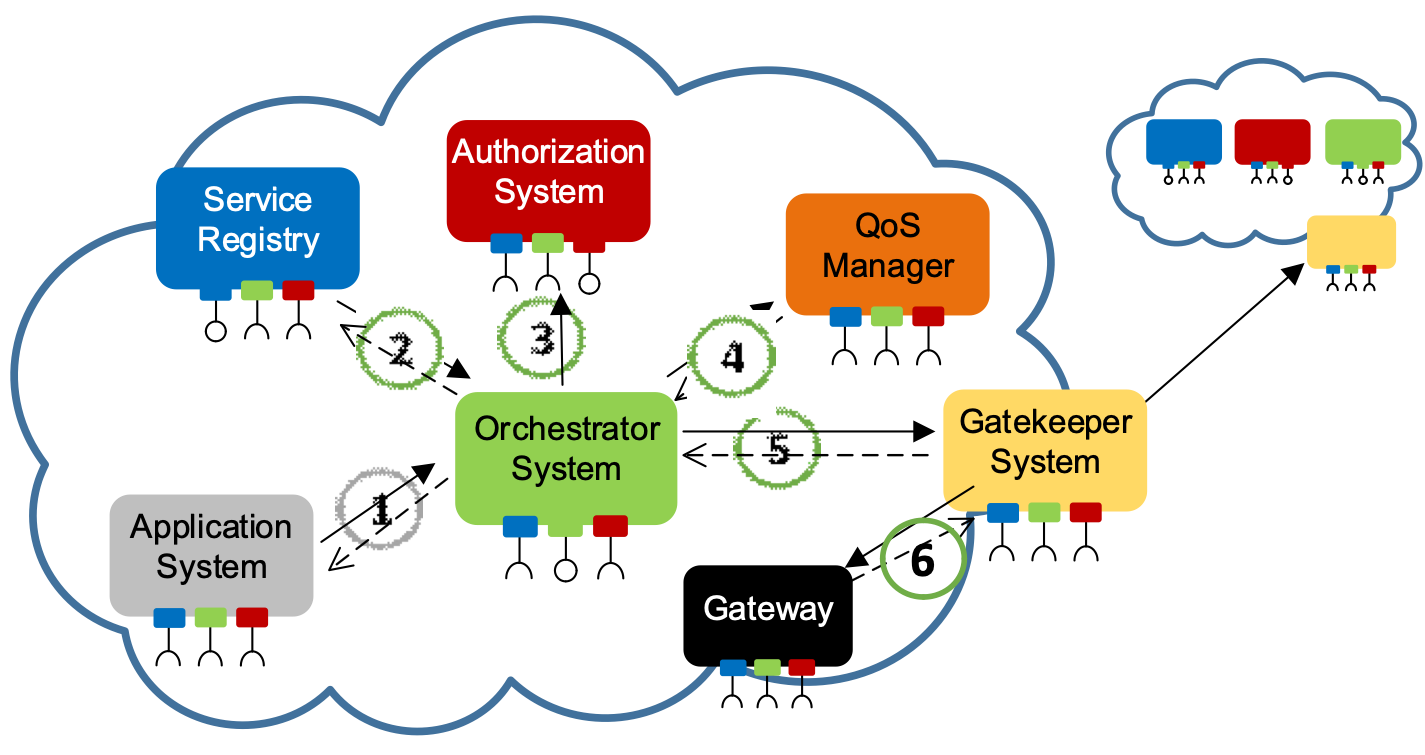
Further details about the inner workings of the Gateway System can be found in [13].

* + 1. **Event Handler**

In the color coding the Event Handler (and its Services) are marked with a turquoise green color. It facilitates communication and data sharing between Application Systems by the way of “event propagation”. It offers two services: event subscription and event publishing. Works in a similar fashion to the MQTT protocol, but it is implemented in REST-JSON, with the help of concurrency handling libraries.

* + 1. **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 5. 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.*

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 autonomously 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.

1. Concepts not covered by G4.0

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. **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.

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.

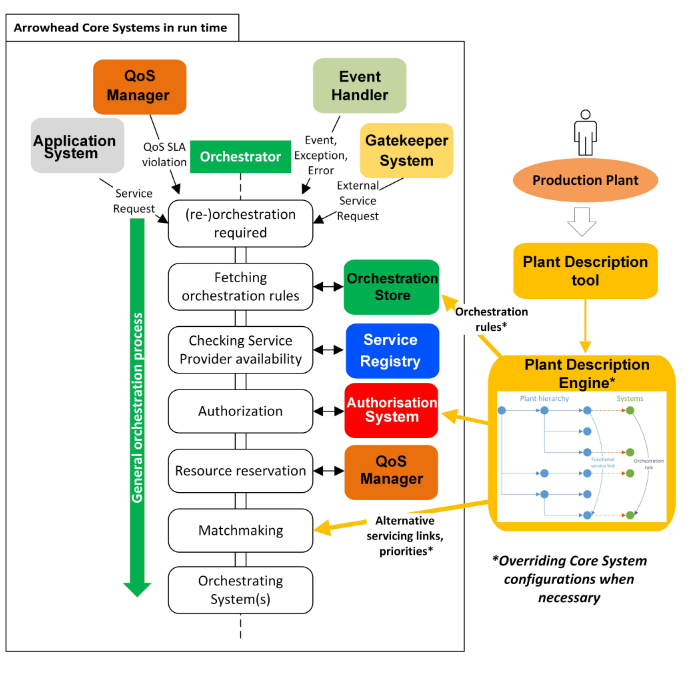
1. **System-of-System Choregraphy**

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 6. 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. Non-functional Requirements

This architecture currently does not have any further requirements.

1. References

[1] Arrowhead ServiceRegistry G4.0 SysDD

[2] Arrowhead ServiceDiscovery Service G4.0 SD

[3] Arrowhead ServiceDiscovery Service G4.0 IDD

[4] Arrowhead Authorization G4.0 SysDD

[7] Arrowhead TokenGeneration Service G4.0 SD

[8] Arrowhead TokenGeneration Service G4.0 IDD

[10] Arrowhead Orchestrator G4.0 SysDD

[11] Arrowhead Orchestration Service G4.0 SD

[12] Arrowhead Orchestration Service G4.0 IDD

[13] Arrowhead Gateway G4.0 SysDD

1. Revision history
   1. **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 |
| 3 | 2018-02-15 | M3 | Updated for M3 | Csaba Hegedűs |
| 4 | 2018-05-23 | G4.0 | Updated for G4.0 | Zoltán Umlauf |

* 1. **Quality Assurance**

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