Document title: Arrowhead Tools Deliverable D3.2

Version	**Status**
1.0	final

Author	**Contact**
Pal Varga, Szvetlin Tanyi	pvarga@tmit.bme.hu

Deliverable D3.2 “Consolidated SOA framework platform architecture, design and sw implementations of core systems – Y1”

Work package leader: Pal Varga
pvarga@tmit.bme.hu

Szvetlin Tanyi
szvetlin@aitia.ai

Daniela Cancila
daniela.cancila@cea.fr

Abstract

This document constitutes deliverable D3.2 of the Arrowhead Tools project.

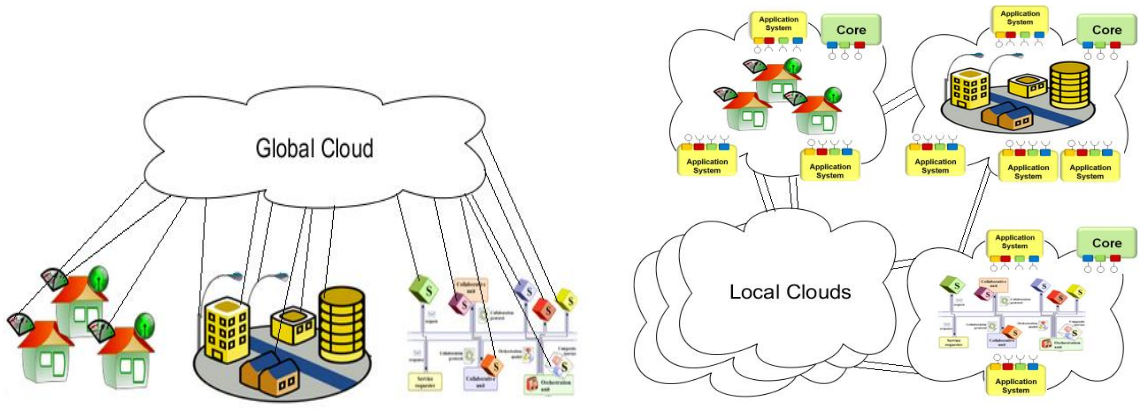
It summarizes the state of the Service Oriented Architecture-based Arrowhead Framework – regarding its platform architecture, design, and software implementations. Furthermore, this document presents the plans for year 1 of the Arrowhead Tools project, regarding the core components of the Arrowhead Framework, as well as some new concepts to be adopted.

Table of contents

1. Introduction

The Arrowhead Framework is addressing IoT based automation. The approach is that the information exchange of IoT's are abstracted to services. This is to enable IoT interoperability in-between almost any IoT's. The creation of automation is based on the idea of Local Clouds. A local Arrowhead cloud can compared to global cloud provide improvements and guarantees regarding:

- Real time data handling
- Data and system security
- Automation system engineering
- Scalability of automation systems

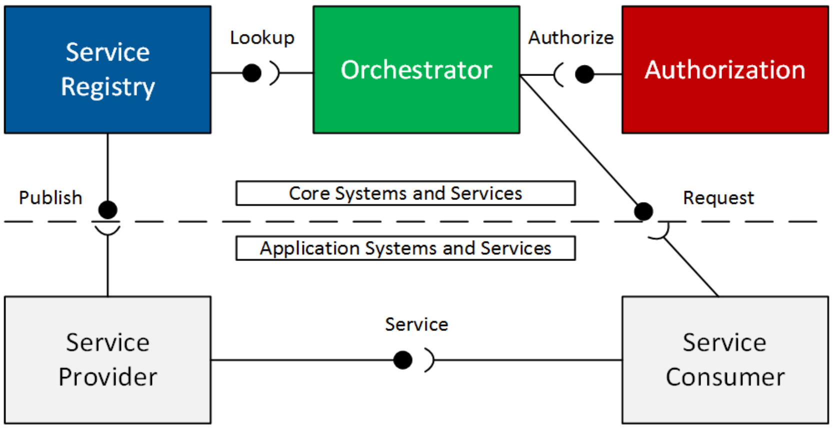


The Arrowhead Framework provides support for building system of systems (SoS) based on service-oriented architecture patterns. Each SoS consists of various Application systems already existing or under development. These Application systems then utilize the Core Systems and their Services that provide support in answering fundamental questions related to governance and operational management, for example:

- How does a service provider system make its possible consumers aware of its available service instance(s)?
- How can a service consumer find (discover) what service instance(s) it might be interested in and allowed to consume?
- How do we remotely control (i.e. orchestrate) which service instances a consumer shall consume?
- How does a service provider determine what consumer(s) to accept?

The smallest unit of governance within the Arrowhead Framework is hence related to a Local Cloud (LC) in general, a closed, local industrial network. Each Local Cloud must at least host the mandatory core systems within its network: creating the minimal the support functionality needed to enable collaboration and information exchange between the various systems within the local cloud. The three mandatory systems for each Local Cloud are:

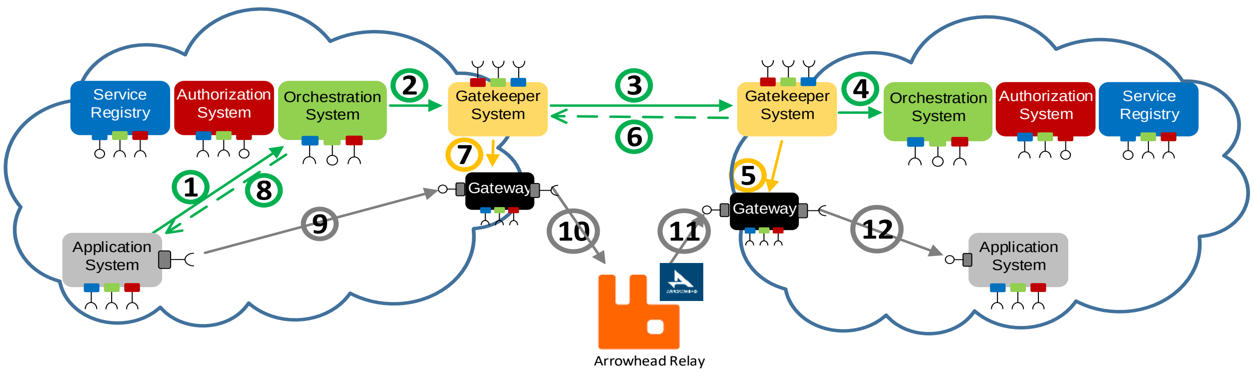
- Service Registry,
- Orchestrator,
- Authorization.



In addition to the mandatory core systems, a number of additional, supporting core systems and services are provided to enable the design, engineering, operation and maintenance of IoT-based automation system of systems. Such supporting core systems are:

- Gatekeeper and Gateway systems
- Event Handler system
- SystemRegistry system
- DeviceRegistry system
- Data Manager system
- Quality of Service (QoS) Manager and Monitor systems
- Translation system
- Plant Description system
- System Configuration system,
- ...and many more.

Inter-cloud information exchange is supported by Gatekeeper (control plane) and Gateway Systems (data plane), whereas security issues are covered through various measures, including AAA functions, certificate handling or data encryption.



2. Arrowhead Framework Mandatory Core – v4.1.3

The complete documentation with proper hyperlinks is publicly available at:

https://github.com/arrowhead-f/core-java-spring

3. Service Registry

3.1 System Design Description Overview

This System provides the database, which stores information related to the currently actively offered Services within the Local Cloud.

The purpose of this System is therefore to allow:

- Application Systems to register what Services they offer at the moment, making this announcement available to other Application Systems on the network.
- They are also allowed to remove or update their entries when it is necessary.
- All Application Systems can utilize the lookup functionality of the Registry to find Public Core System Service offerings in the network, otherwise the Orchestrator has to be used.

However, it is worth noting, that within this generation the lookup functionality of Services is integrated within the "orchestration process". Therefore, in the primary scenario, when an Application System is looking for a Service to consume, it shall ask the Orchestrator System via the Orchestration Service to locate one or more suitable Service Providers and help establish the connection based on metadata submitted in the request. Direct lookups from Application Systems within the network is not advised in this generation, due to security reasons.

However, the lookup of other Application Systems and Services directly is not within the primary use, since access will not be given without the Authorization JWT (JSON Web Token). The use of the TokenGeneration is restricted to the Orchestrator for general System accountability reasons.

3.2 Services and Use Cases

This System only provides one Core Service the **Service Discovery**

There are two use case scenarios connected to the Service Registry.

- Service registration, de-registration
- Service Registry querying (lookup)

The **register** method is used to register services. The services will contain various metadata as well as a physical endpoint. The various parameters are representing the endpoint information that should be registered.

The **unregister** method is used to unregister service instances that were previously registered in the Registry. The instance parameter is representing the endpoint information that should be removed.

The **query** method is used to find and translate a symbolic service name into a physical endpoint, for example an IP address and a port. The query parameter is used to request a subset of all the registered services fulfilling the demand of the user of the service. The returned listing contains service endpoints that have been fulfilling the query.

There is another functionality that does not bound to any Services, just an internal part of the Service Registry. There are two optional cleanup tasks within the Service Registry, which can be used to remove old, inactive service offerings. The first task is based on pinging the service provider and if the provider does not respond to the ping, its offered services will be deleted. The second task is based on a feature, called "Time to Live". Service providers upon registration can provide a timestamp called "end_of_validity" number, which specifies how long the service will be offered by the provider, making the service de-registrations unnecessary, if this task is active. The task is used to remove expired services. The third task is using a feature called "Heartbeat" (Not yet implemented), where the Service provider periodically signals to the Service Registry that it is still alive. When it misses it will be removed. All of these internal tasks can be configured in the application.properties file.

3.3 Security

This System can be secured via the HTTPS protocol. If it is started in secure mode, it verifies whether the Application System possesses a proper X.509 identity certificate and whether that certificate is Arrowhead compliant in its making. This certificate structure and creation guidelines ensure:

- Application System is properly bootstrapped into the Local Cloud
- The Application System indeed belongs to this Local Cloud
- The Application System then automatically has the right to register its Services in the Registry.

If these criteria are met, the Application System's registration or removal message is processed. An Application System can only delete or alter entries that contain the Application System as the Service Provider in the entry.

3.4 Endpoints

The Service Registry offers three types of endpoints. Client, Management and Private.

Swagger API documentation is available on: `https://<host>:<port>`
The base URL for the requests: `http://<host>:<port>/serviceregistry`

Detailed documentation of the endpoints can be found at:

https://github.com/arrowhead-f/core-java-spring/tree/documentation#serviceregistry_endpoints

4. Authorization

4.1 System Design Description Overview

This System has:

- A database that describes which Application System can consume what Services from which Application Systems (Intra-Cloud access rules)
- A database that describes which other Local Clouds are allowed to consume what Services from this Cloud (Inter-Cloud authorization rules)

The purpose of this System is therefore to:

- Provide AuthorizationControl Service (both intra- and inter-Cloud)
- Provide a TokenGeneration Service for allowing session control within the Local Cloud

The purpose of the TokenGeneration functionality is to create session control functionality through the Core Sytems. The output is JSON Web Token that validates the Service Consumer system when it will try to access the Service from another Application System (Service Provider). This Token shall be primarily generated during the orchestration process and only released to the Service Consumer when all affected Core Systems are notified and agreed to the to-be-established Service connection.

This System (in line with all core Systems) utilizes the X.509 certificate Common Name naming convention in order to work.

4.2 Services and Use Cases

This System only provides two Core Services:

- AuthorizationControl
- TokenGeneration

There are two use cases connected to the Authorization System:

- Check access rights (invoke the AuthorizationControl)
- Generate an access token (the Orchestrator invokes the TokenGeneration)

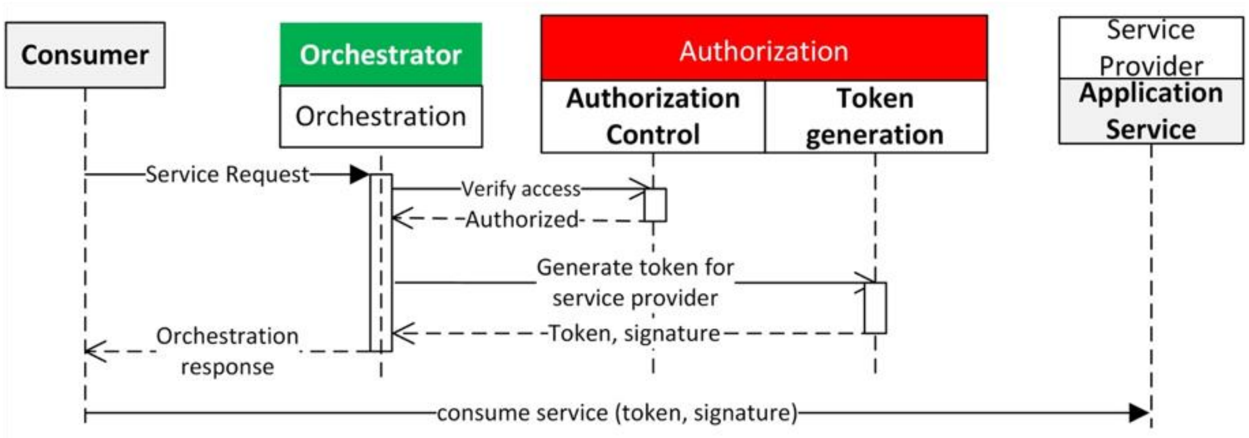


Figure 1. Authorization crosscheck during orchestration process

4.3 Service Description Overview

The AuthorizationControl Service provides 2 different interfaces to look up authorization rights:

- Intra-Cloud authorization: defines an authorization right between a consumer and provider system in the same Local Cloud for a specific Service.
- Inter-Cloud authorization: defines an authorization right for an external Cloud to consume a specific Service from the Local Cloud.

4.4 Endpoints

The Authorization offers three types of endpoints. Client, Management and Private.

Detailed documentation of the endpoints can be found at:

https://github.com/arrowhead-f/core-java-spring#authorization_endpoints

5. Orchestrator

5.1 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.

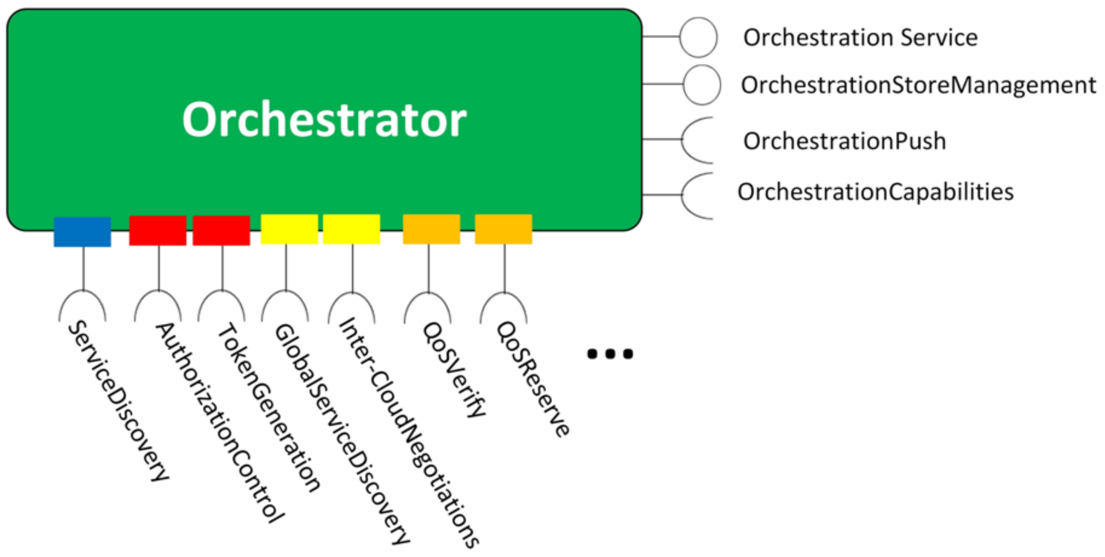
This orchestration rule information can reach the given Application System (consumer) in two different ways: the System itself can request it ("pull") or the Orchestrator itself can update the System when it is needed ("push method"). However, in both cases, there shall be an underlying, hidden process ("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. This figure depicts the mandatory and optional interfaces of this 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 fulfil 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.

Orchestrator can be used in two ways. The first one uses predefined rules (coming from the Orchestrator Store DB) to find the appropriate providers for the consumer. The second option is the dynamic orchestration in which case the core service searches the whole local cloud (and maybe some other clouds) to find matching providers.

5.1.1 Store Orchestration:

- requester system is mandatory,
- requested service and all the other parameters are optional,
- if requested service is not specified, then this service returns the top priority local provider of all services contained by the orchestrator store database for the requester system. if requested service is specified, then you have to define the service definition and exactly one interface (all other service requirements are optional). In this case, it returns all accessible providers from the orchestrator store database that provides the specified service via the specified interface to the specified consumer.

5.1.2 Dynamic Orchestration:

- requester system is mandatory,
- requested service is mandatory, but just the service definition part, all other parameters of the requested service are optional,
- all other parameters are optional

5.1.3 Orchestration flags:

- `matchmaking`: the service automatically selects exactly one provider from the appropriate providers (if any),
- `metadataSearch`: query in the Service Registry uses metadata filtering,
- `onlyPreferred`: the service filters the results with the specified provider list,
- `pingProviders`: the service checks whether the returning providers are online and remove the unaccessible ones from the results,

- `overrideStore`: Services uses dynamic orchestration if this flag is true, otherwise it uses the orchestration store,
- `enableInterCloud`: the service can search another clouds for providers if none of the local cloud providers match the requirements,
- `triggerInterCloud`: the service skipped the search in the local cloud and tries to find providers in other clouds instead.

5.2 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.

Use case 1: *Service Request From Application System*

Name	Description
ID	Orchestration 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	- The Application System requests orchestration. - The Orchestrator System begins the orchestration process with the other Core Systems. - The Orchestrator System responds to the Application System based on the request.
Postconditions	-

Use case 2: *Orchestration information pushed to Application System*

Name	Description
ID	Orchestration Push
Brief	The Orchestrator pushes new information on Application Systems

Name	Description
Description	
Primary Actors	Orchestrator
Secondary Actors	the other Core Systems instances of the Local Cloud
Preconditions	Change in the Orchestration Store.
Main flow	- The Orchestrator detects a change in the Orchestration Store. - The Orchestrator begins the orchestration process with the other Core Systems for every change in the Store. - The orchestrator pushes new connection rules to the Application Systems based on the new Store entry.
Postconditions	-

Use case 3: *Orchestration information pushed to Application System*

Name	Description
ID	Orchestration 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	Change in the Orchestration Store.
Main flow	- The Orchestrator detects a change in the Orchestration Store. - The Orchestrator begins the orchestration process with the other Core Systems for every change in the Store. - The orchestrator pushes new connection rules to the Application Systems based on the new Store entry.
Postconditions	-

5.3 Endpoints

The Orchestrator offers three types of endpoints. Client, Management and Private.

Detailed documentation of the endpoints can be found at:

https://github.com/arrowhead-f/core-java-spring#orchestrator_endpoints

6. Gatekeeper System

This System provides two Core Services:

- Global Service Discovery (GSD)
- Inter-Cloud Negotiations (ICN)

These Services are part of the inter-Cloud orchestration process and peferably these Services are not available for Application Systems.

Note: Self-orchestrating[1] Application Systems are currently not in scope, however they are not excluded from the Arrowhead framework. They are supported at high-level design, implementations have not been published yet. Therefore, the internal Services of the Gatekeepers are not public for Application Systems, hence not documented as such.

The first is the Global Service Discovery (GSD) process, which aims at locating adequate service offerings in neighboring Clouds. The second is the Inter-Cloud Negotiations (ICN) process, in which mutual trust is established between two Clouds and the actual connection between endpoints is then built up. Working together with the Orchestrators of both Clouds, at the end a servicing instace can be created.

These Services each have two sets of interfaces:

- Provided intra-Cloud to the Orchestrator
- Provided inter-Cloud among the Gatekeepers

7. Gateway System

This System provides two Core Services:

- Session Establish
- Session Management

The Session Establish Service has two interfaces:

- Connect to Consumer
- Connect to Provider

These Services are part of the inter-Cloud orchestration process.
Main flow:

1- The Gatekeeper sends a ConnectToConsumerRequest to the Gateway.

2- The Gateway internally adds a new ActiveSession object to the activeSessions HashMap.

3- The Gateway starts a new thread (secure/insecure based on connection mode).

4- The Gateway sends a ConnectToConsumerResponse to the Gatekeeper.

In the thread:

5- The Gateway creates a serverSocket/sslServerSocket.

[1] Self-orchestrating Application Systems implement all Core Service interfaces and capable of running an orchestration process on their own – without the use of the central Orchestrator.

6 – The Consumer connects to the port of the serverSocket.

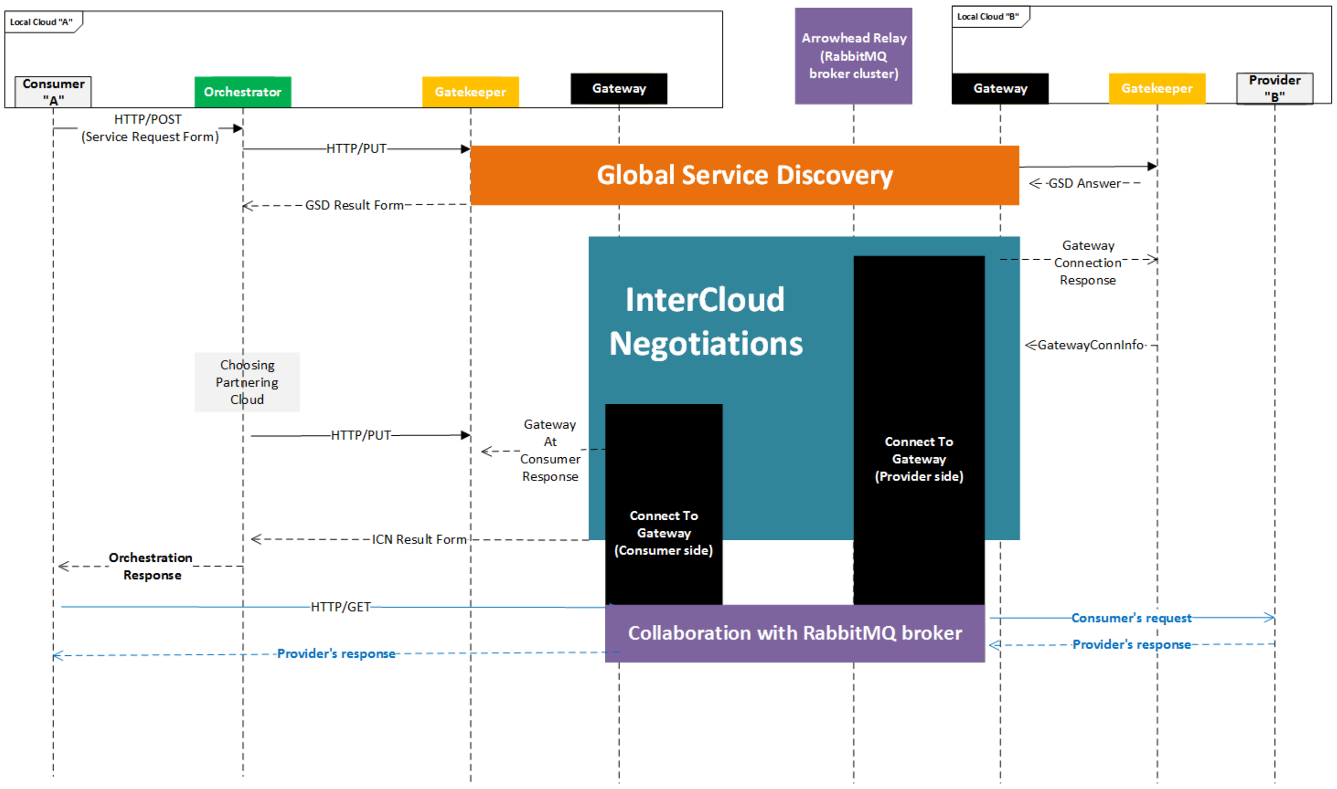
7 - The Gateway accepts the connection and creates a socket/SSLSocket for the Consumer.

8 – The Gateway gets the request from the Consumer through the socket/SSLSocket and forwards it to the Broker.

9- The Gateway gets the response from the Provider through the Broker and forwards it to the Consumer through the socket towards the consumer.

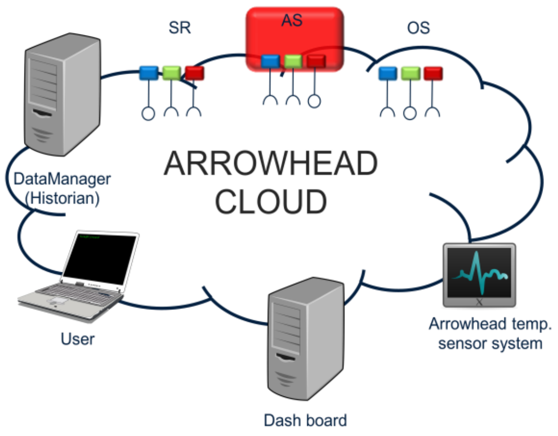
10- The Gateway checks the control messages from Broker.

11- Repeat the steps 8-10. until one of the sockets gets closed or it gets a "close" message from Broker via "controlQueue".



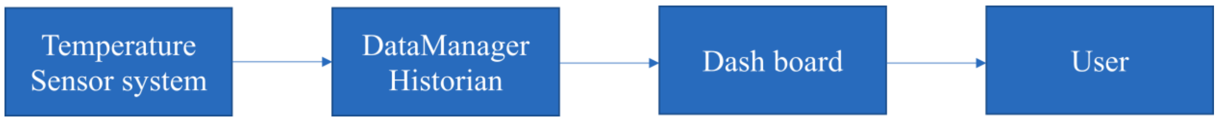
8. DataManager

The DataManager is a system that provides services for storing and retrieving sensor data. The two provided services are Proxy and Historian.



The Proxy service can cache messages from battery-powered systems such as wireless IoT devices. The Historian service provides features for storing large amounts of sensor data for later retrieval and export. The Historian can be used for usage in dashboards, backup of data and similar use cases. An example scenario is the following:

- A temperature providing Arrowhead System is using Orchestration to push sensor data to the Historian service of the DataManager,
- A user uses a web-based dash board to visualize the temperature,
- The Dash board fetches data in real-time from the Historian.



Currently, the status of the DataManager is that it is ready for testing before its' development branch is pulled into the main branch of the Arrowhead Framework. It has not yet been moved into the Spring based framework. This is planned to be implemented during year 1.

9. Extended Historian System

With regard to production processes, systems for evaluation of process data can contribute to improved control strategies and decision support for human intervention. Keywords are predictive control, machine learning, statistical analysis or neuronal networks. All those technologies have in common that they may use time series of process data as raw data input. An Arrowhead core system should support data acquisition and management.
There exists a Historian system, which could take over this role. But it is limited in functionality. Thus IFAK will develop an Extended Historian system as one of the optional Arrowhead automation support core systems. Compared to the existing Historian system it will contain:

- *Functionality for scheduling of data acquisition processes*
  So the data acquisition is triggered by the Extended Historian system, which makes it possible to gather data from different sources at equidistant and more or less synchronized time points.
- *A set of interfaces for connection of data analysis tools*
  This makes it possible to use the data from different data analysis tools (candidates are

GNU R, Python (with SciPy, scikit-learn, matplotlib) or MATLAB), which possibly don't implement any of the Arrowhead Framework interfaces.

- *A set of interfaces for non-Arrowhead services*
  So data of an Arrowhead local cloud can be managed together with data from outside of that communication system. It has special value for legacy data sources, which currently don't support the Arrowhead Framework.

This Extended Historian system makes it unnecessary in many cases to introduce proprietary data acquisition systems. Data analytics scenarios can easily be implemented by using an automation system based on the Arrowhead Framework and a common data analytics tool.
Currently the specification of the system is work in progress. It specifies use cases, software structures and high-level descriptions of the necessary interfaces.

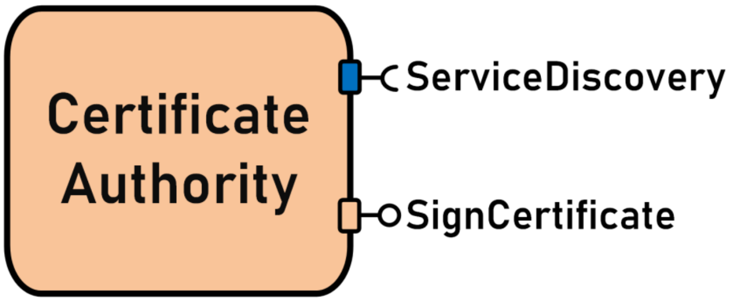
10. Translator

The Translator system is responsible for translating between different communication and semantic protocols. The Translator provides services for configuring translation tasks and translation between protocols such as HTTP and OPC-UA or sematic formats like JSON and XML,

The current development status is that translation between Arrowhead-compliant communication and FIWARE HTTP and CoAP is planned to be supported by end of Q1 2020. The Translator must also be converted to the new Spring-based Arrowhead Framework software base.

11. Certificate Authority

The purpose of Certificate Authority (CA) system is signing any descendant certificate of the systems in an Arrowhead Local Cloud (LC). All parties in an Arrowhead Local Cloud must trust the CA inside the related LC. The certificate of the CA may be signed by a central authority (e.g. Arrowhead Consortium), so, the chain of trust can be established allowing different LCs to interconnect with each other in secure manner.



The Certificate Authority system consumes ServiceDiscovery service and provides SignCertificate service.

The CA provides signed certificate for a requester entity (application system) inside a LC

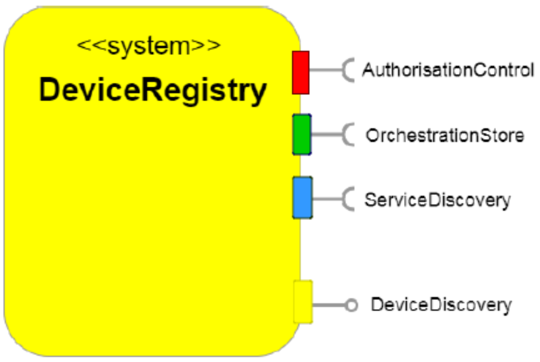
1. the requester entity has to construct a CSR and send it to the CA
2. the CA verifies the signature inside the CSR
3. if the signature verification is successful then the CA generates and sends back a signed certificate for the requester entity.
4. using this certificate, the requester entity is able to communicate in secure manner with the systems inside the LC

At the time this document is written the CA system is involved into the Arrowhead Onboarding procedure only. For the onboarding procedure use cases please refer to the following sections

12. Device Registry

The DeviceRegistry system stores information and unique identities for devices registered within an Arrowhead local cloud. The registration into a local cloud is part of the bootstrapping process of a local cloud. This registry shall in addition to registering device identity, also store metadata about the device and shall hold data on systems that are deployed to each registered device.

The DeviceRegistry system is one of the support core systems of the Arrowhead Framework. It provides one service and consumes the mandatory core services, see the below figure.



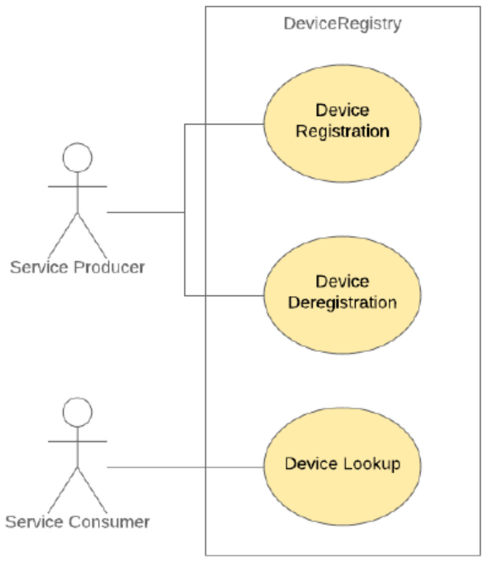
This registry in combination with the SystemRegistry creates a chain of trust from a hardware device to a hosted software system and its associated services.

The DeviceRegistry provides DeviceDiscovery service for device registration, de-registration and lookup based on a Java-based REST interface, which uses a MySQL database. It consumes the three mandatory core services, AuthorisationControl, OrchestrationStore and ServiceDiscovery.

All Arrowhead Framework Devices within a local cloud shall register within the DeviceRegistry by using the DeviceDiscovery service. As such DeviceDiscovery is regarded as a well known service, and shall be accessible using a multitude of SOA protocols like e.g. REST, CoAP, MQTT. The current implemented protocols for the DeviceRegistry system are documented 19nt he IDD document.

This registry in combination with the SystemRegistry is necessary to create a chain of trust from a hardware device to a hosted software system and its associated services.

The actors of typical use-cases are represented by the following figure.



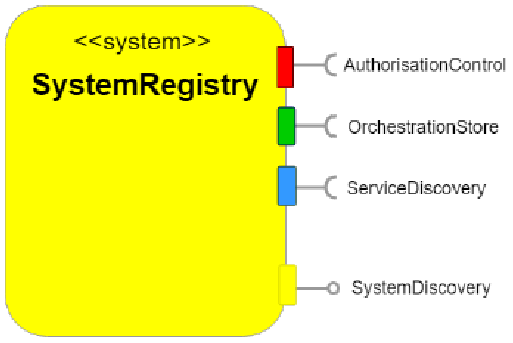
Device Lookup The use case Device Lookup is used by the service consumer to get the list of registered devices 20nt he DeviceRegistry system, e.g. searches for a specific device instance with a specific ID (primary key).

Device Registration The use case Device Registration is used by the service producer to register its device 20nt he DeviceRegistry system, e.g. stores a device instance 20nt he database (instances must not contain Ids, but embedded entities can have Ids if they already exist).

Device De-registration The use case Device De-registration is used by the service producer to delete its device from the DeviceRegistry system, e.g. removes a device instance from the database.

13. System Registry

The SystemRegistry system is used to provide a local cloud storage holding the information on which systems are registered in a local cloud, meta-data of these registered systems and the services these systems are designed to consume. The SystemRegistry system holds for the local cloud unique system identities for systems deployed within the Arrowhead Framework local cloud.

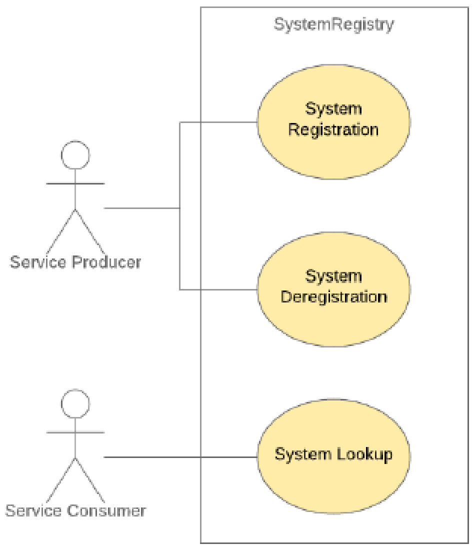


The SystemRegistry provides SystemDiscovery service for system registration, de-registration and lookup based on a Java-based REST interface, which uses a MySQL database. It consumes the three mandatory core services, AuthorisationControl, OrchestrationStore and ServiceDiscovery.

All Arrowhead Framework Systems within a local cloud shall register within the SystemRegistry by using the SystemDiscovery service. As such SystemDiscovery is regarded as a well known service, and shall be accesible using a multitude of SOA protocols like e.g. REST, CoAP, MQTT. The current implemented protocols for the SystemRegistry system are documented 21nt he IDD document.

This registry in combination with the DeviceRegistry is necessary to create a chain of trust from a hardware device to a hosted software system and its associated services.

The actors of typical use-cases are represented by the following figure.



System Lookup The use case System Lookup is used by the service consumer to get the list of registered systems 21nt he SystemRegistry system, e.g. searches for a specific system instance with a specific ID (primary key).

System Registration The use case System Registration is used by the service producer to register its system 21nt he SystemRegistry system, e.g. stores a system instance 21nt he database (instances must not contain Ids, but embedded entities can have Ids if they already exist).

System De-registration The use case System De-registration is used by the service producer to delete its system from the SystemRegistry system, e.g. removes a system instance from the database.

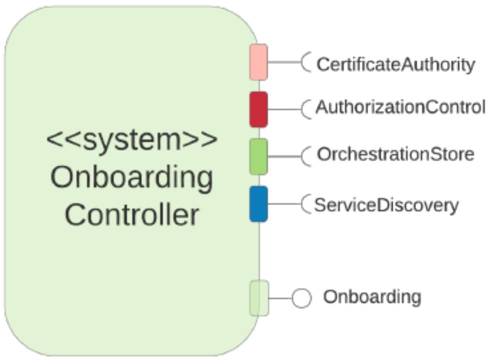
14. Onboarding Controller

Onboarding Controller system:

- A system at the edge of the Arrowhead local cloud, which is not part of the local cloud chain of trust
- It accepts all devices to connect via the Onboarding service, thus it is the first entry point to the local cloud
- It has a certificate for the https communication with the device
- (Optionally) the certificate is provided by a public CA (e.g. verisign)

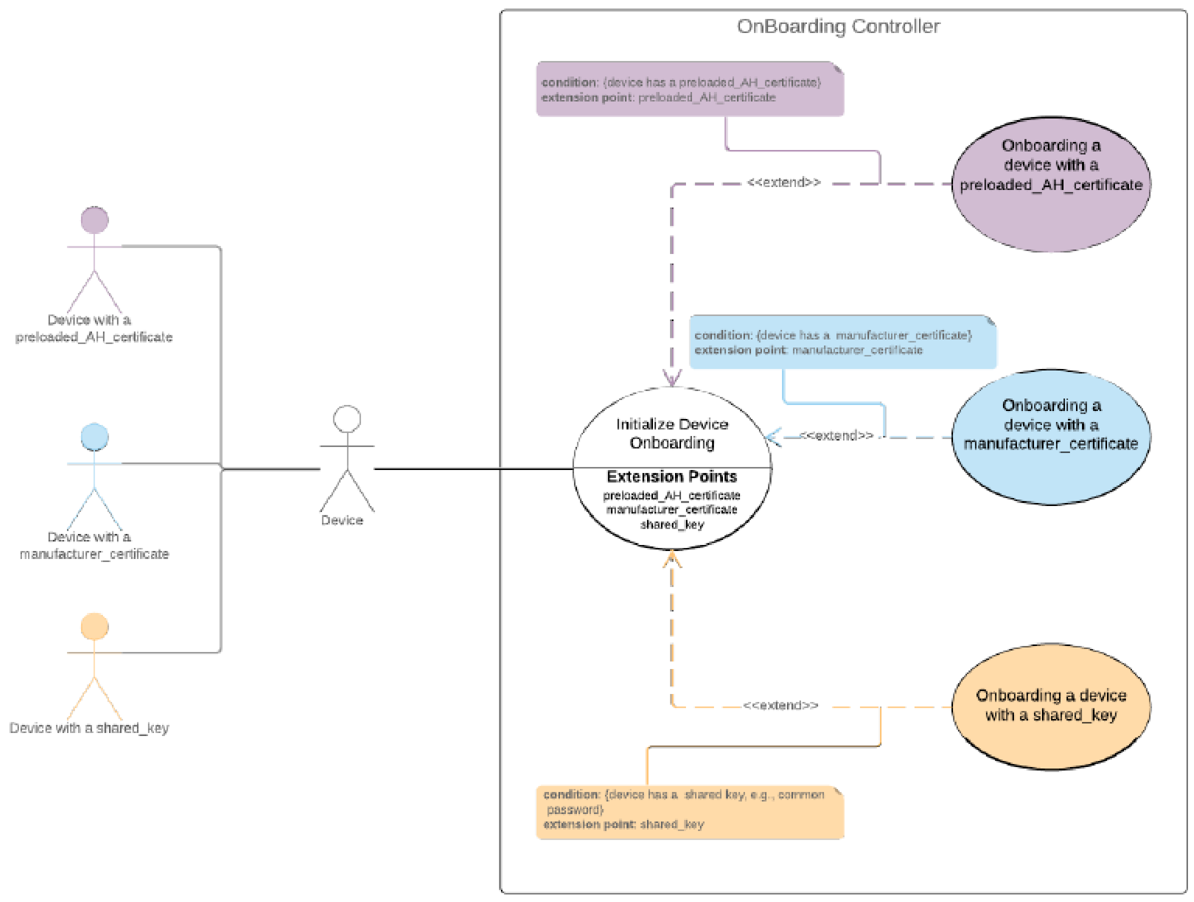
On success, the system provides:

- the endpoints of the DeviceRegistry, SystemRegistry and ServiceRegistry systems
- the Arrowhead CA certificate
- an Arrowhead CA issued „onboarding" certificate



A typical use-case scenario is the following.

The actors can be devices with different credentials.


OnBoarding Controller
condition: {device has a preloaded_AH_certificate}
extension point: preloaded_AH_certificate
<<extend>>
Onboarding a device with a preloaded_AH_certificate
Device with a preloaded_AH_certificate
condition: {device has a manufacturer_certificate}
extension point: manufacturer_certificate
Onboarding a device with a manufacturer_certificate
<<extend>>
Initialize Device Onboarding
Extension Points
preloaded_AH_certificate
manufacturer_certificate
shared_key
Device with a manufacturer_certificate
Device
Device with a shared_key
<<extend>>
Onboarding a device with a shared_key
condition: {device has a shared key, e.g., common password}
extension point: shared_key

The onboarding procedure is needed when a new device produced by any vendor (e.g. Siemens, Infineon, Bosch, etc.), containing a security controller (e.g. TPM), wants to interact with the Arrowhead local cloud. To assure that the cloud is not compromised upon the arrival of this new device, it is important to establish a chain of trust from the new hardware device, to its hosted application systems and their services. Thus, the onboarding procedure makes possible that the device, systems and services are authenticated and authorized to connect to the Arrowhead local cloud.

The use cases in which the external actor interacts with the Arrowhead local cloud during onboarding include:

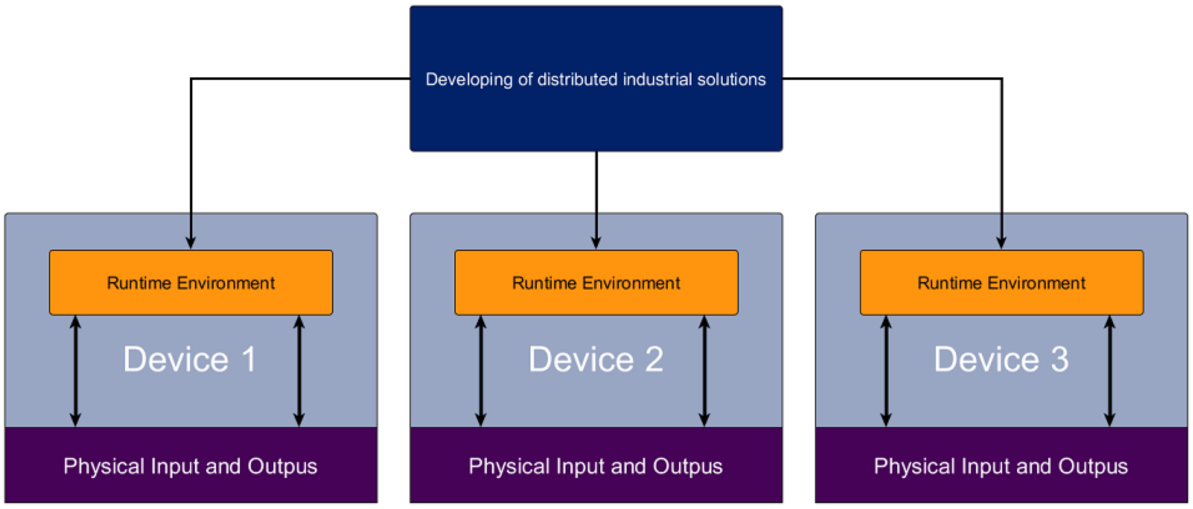
- Initialize Device Onboarding (via the Onboarding Controller system)
- Register a Device 23nt he DeviceRegistry (via the DeviceRegistry system)
- Register a System 23nt he SystemRegistry (via the SystemRegistry system)
- Register a Service 23nt he ServiceRegistry (via the ServiceRegistry system)
- Start normal operation (e.g., service lookup, service consumption, etc.)

15.How to use 4DIAC's Arrowhead library

This subsection first introduces the main concepts behind the 4DIAC Eclipse project, then it shows its conceptual as well as implementation-specific integration with the Arrowhead Framework.

An application developed using the IEC 61499 standard uses Function Blocks (FB). There are ways of creating and defining FBs, 24nt he some point 24nt he workflow ranging from FBs to actually controlling something physical, the logic behind the FBs and the FB networks has to be implemented. That's where the runtime environment comes in. This software loads the network of FBs and then executes the events and follows the rules of the standard like the one seen here.

So, where is the runtime environment 24nt he standard? Nowhere directly. For executing the distributed control applications modeled with IEC 61499 the standard defines the device model containing resources, the FB execution model, and the management model allowing tools to configure devices. This is what a runtime environment provides. 24nt he next image, the idea behind decoupling the application development and the execution is presented.


Developing of distributed industrial solutions
Runtime Environment
Device 1
Physical Input and Outpus
Runtime Environment
Device 2
Physical Input and Outpus
Runtime Environment
Device 3
Physical Input and Outpus

The rectangle 24nt he top represents the system according to IEC 61499. Here, you need a tool that allows creating FBs and connecting them. Also, it should represent the devices of your system, and some method to show the part of your application which will be deployed to each device. This is usually done on a normal computer.

The big squares 24nt he second row represent real devices such as PLCs, control hardware, or a Raspberry Pi. In these devices, the mentioned runtime environment needs 24nt he. It receives information from the top rectangle to create the network of FBs, execute FBs, send events between FBs, and so on. The devices normally have inputs and outputs which are accessed by the runtime environment.

Eclipse 4diac provides two main components for developing and executing distributed control systems compliant to the IEC 61499 standard:

- 4diac FORTE is a small portable C++ implementation of an IEC 61499 runtime environment which supports executing IEC 61499 FB networks on small embedded devices. 4diac FORTE typically runs on top of a (real-time) OS. 4diac FORTE is a multi-threaded and low memory-consuming runtime environment. It has been tested on several different operating systems, for example, Windows, Linux (i386, amd64, ppc, xScale, arm), NetOS, eCos, rcX from Hilscher, vxWorks, and freeRTOS.
- 4diac IDE: is an integrated development environment written in Java, based on the Eclipse framework. It provides an extensible engineering environment for 25nt he25ur distributed control applications compliant to the IEC 61499 standard. You use 4diac IDE to create FBs, applications, configure the devices and other tasks related to IEC 61499. Within 4diac IDE, these results can also be deployed to devices running 4diac FORTE or other compliant run-time environments.

Regarding the Arrowhead Framework integration, 4DIAC provides means for applications to communicate with the core components of the AF. These are implemented through Function Blocks.

The basic sequence for having a service being produced and consumed 25nt he AF is the following:

1. Configure your local cloud: First, you need to tell the Authorization System which services are allowed to be consumed by which system and to the Orchestrator System where are the sevice providers for the consumers (IP and Port).
2. When the service provider is connected to the local cloud, it should register to the Service Registry.
3. The service consumer will do the same. In addition, it will contact the Orchestrator asking for the endpoint of the service provider.
4. The Orchestrator will look in its private database (configured in point 1) for the 25nt he25u of the service provider, and check with the Authorization System if the consumer is allowed to consume from the provider.
5. If yes, the Orchestrator answers the consumer (point 3) with the endpoint of the service provider.
6. The service consumer connects to the service provider using the endpoint given in point 5, and consumes the services.

As seen before, the service interface between the consumer and provider is not fixed and the AF has nothing to do with it. The AF provides the support to register services, and look for them.

The developed 4DIAC's Arrowhead library provides the FBs and AF types to communicate to the official Arrowhead Framework using HTTP/JSON.

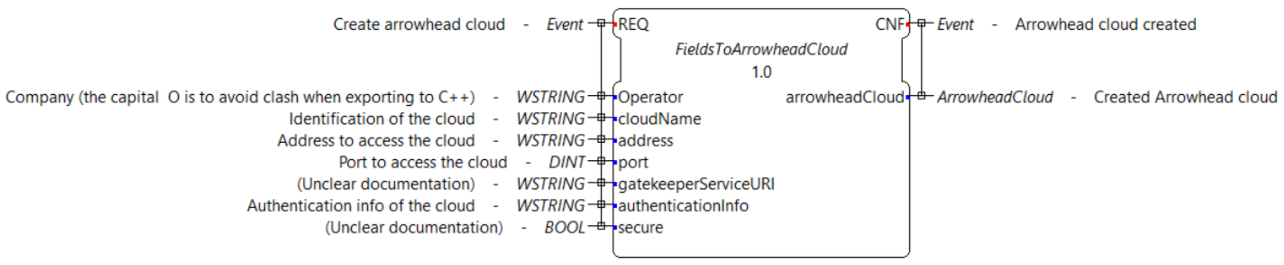
15.1 Enabling the Arrowhead Module in 4diac FORTE

The first thing that's needed is to have a version of 4diac FORTE with the Arrowhead Module enabled. 25nt he25u, you'll need to compile your own 4diac FORTE. To do that, follow these steps and in Cmake set the variable FORTE_MODULE_Arrowhead and FORTE_COM_HTTP to TRUE.

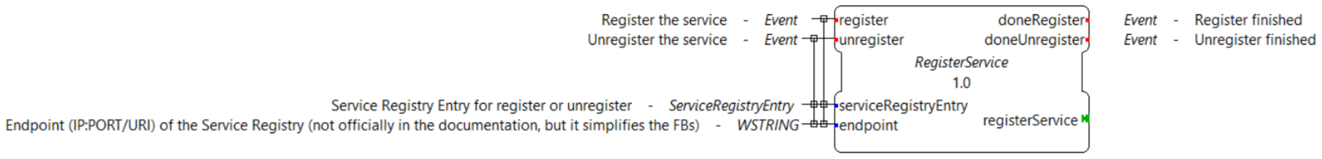
After 4diac FORTE compiles, you'll be ready to use the FBs library in 4diac IDE.

15.2 FBs in 4diac IDE

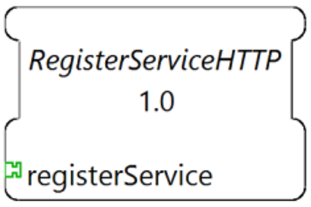
The 4diac IDE is where the library of Function Blocks were defined, and it was done in three levels. 26nt he lowest level, helper Function Blocks allow the user to create the Arrowhead Framework types using standard types from IEC 61499. The following figure shows the Function Block to create an Arrowhead Cloud type, which can later be connected to the upper levels.


Create arrowhead cloud - Event
REQ
FieldsToArrowheadCloud
1.0
CNF
Event - Arrowhead cloud created
Company (the capital O is to avoid clash when exporting to C++) - WSTRING
Operator
Identification of the cloud - WSTRING
cloudName
Address to access the cloud - WSTRING
address
Port to access the cloud - DINT
port
(Unclear documentation) - WSTRING
gatekeeperServiceURI
Authentication info of the cloud - WSTRING
authenticationInfo
(Unclear documentation) - BOOL
secure
arrowheadCloud
ArrowheadCloud - Created Arrowhead cloud

26nt he second level, the actual Arrowhead services were implemented. These Function Blocks offer an adapter following the IEC 61499 standard, in order to decouple the abstract definition of services from the actual implementation. As an example, the following figure shows the Function Block to register and unregister a service 26nt he Service Registry. The data inputs are a Service Registry Entry type (protocol independent) and the endpoint to connect to. The "registerService" adapter 26nt he below right side, offers a plug to the actual implementation of the communication, passing all needed data.

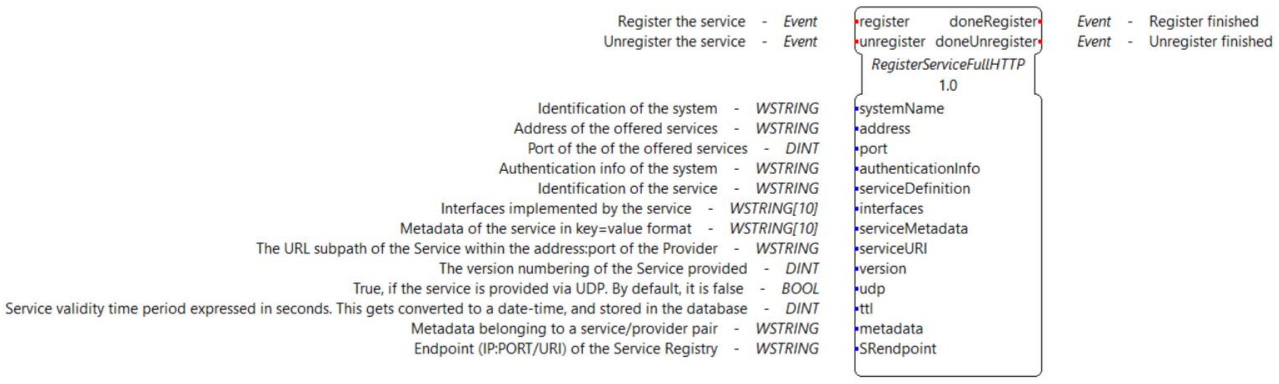

Register the service - Event
register
Unregister the service - Event
unregister
doneRegister
doneUnregister
Event - Register finished
Event - Unregister finished
RegisterService
1.0
Service Registry Entry for register or unregister - ServiceRegistryEntry
serviceRegistryEntry
Endpoint (IP:PORT/URI) of the Service Registry (not officially in the documentation, but it simplifies the FBs) - WSTRING
endpoint
registerService

The socket for this adapter is given by each implementation. The next figure shows the Function Block that implements the actual HTTP communication to the Service Registry. Its only input is the socket adapter, the counter part of the plug adapter from the previous figure. By connecting them, the information is passed to this Function Block that handles the communication.



This decoupled architecture allows to quickly implement another type of communication. For example, if the Service Registry offers an OPC UA interface, only the Function 26nt he26u the last figure must be re-implemented with the specifics of OPC UA.

27nt he top most level, sub-applications were implemented in order to facilitate the user using the library. For example, for registering a service, instead of building from the lowest level, which requires many Function Blocks and connections, an encapsulated sub-application is provided that offers all parameters for it. The following figure shows the sub-application with all parameters to register and unregister a service.


Register the service - Event
Unregister the service - Event
register
doneRegister
unregister
doneUnregister
Event - Register finished
Event - Unregister finished
RegisterServiceFullHTTP
1.0
Identification of the system - WSTRING
Address of the offered services - WSTRING
Port of the of the offered services - DINT
Authentication info of the system - WSTRING
Identification of the service - WSTRING
Interfaces implemented by the service - WSTRING[10]
Metadata of the service in key=value format - WSTRING[10]
The URL subpath of the Service within the address:port of the Provider - WSTRING
The version numbering of the Service provided - DINT
True, if the service is provided via UDP. By default, it is false - BOOL
Service validity time period expressed in seconds. This gets converted to a date-time, and stored in the database - DINT
Metadata belonging to a service/provider pair - WSTRING
Endpoint (IP:PORT/URI) of the Service Registry - WSTRING
systemName
address
port
authenticationInfo
serviceDefinition
interfaces
serviceMetadata
serviceURI
version
udp
ttl
metadata
SRendpoint

15.3 **Purpose of available FBs in the Arrowhead Library**

The following gives informatoin about the purposse of all FBs and SubApps 27nt he library. The parameters of each one of them is documented 27nt he FBs and SubApps itself.

The ones marked in bold are the top level, and usually, the only ones that you need in order to quickly use the library.

- Common
  - FieldsToArrowheadCloud: Translate individual fields to an ArrowheadCloud type
  - FieldsToArrowheadService: Translate individual fields to an ArrowheadService type
  - FieldsToArrowheadSystem: Translate individual fields to an ArrowheadSystem type
  - JSON
    - ANYToJSON: Transform 61499 types to JSON format
    - GetArrayResponseFromJSON: Transform a HTTP response to an array
- Service Registry
  - FieldsToServiceQueryForm: Translate individual fields to a Service Query Form type
  - FieldsToServiceRegistryEntry: Translate individual fields to a Service Registry type
  - GetEndpointFromServiceRegistry: Get the IP:PORT/URI endpoint from a Service Registry Entry
  - QueryService: Query Service Function Block
  - QueryServicesAdp: Query services adapter
  - RegisterService: Register Service Function Block
  - RegisterServiceAdp: Register Service Adapter
  - ServiceRegistryEntry2ServiceRegistryEntry: Helper FB to set the connection to a Service Registry Entry type

- HTTP
  - QueryServiceHTTP: Query for Services using HTTP
  - QueryServiceHTTPSub: Query services using HTTP with the service defined
  - QueryServiceHTTPSubFull: Query services using HTTP with all service’s fields to be defined
  - RegisterMultipleServicesHTTP: Register many services with different serviceDefinition and serviceURI
  - RegisterServiceFullHTTP: Register a Service using HTTP. All possible parameters are available to be set
  - RegisterServiceHTTP: Register Service using HTTP
  - RegisterServicePartialHTTP: Register a Service using HTTP. The system information is encapsulated

- Orchestrator
  - FieldsPreferredProvider: Translate individual fields to a PreferredProvider type
  - FieldsToServiceRequestForm: Translate individual fields to a ServiceRequestForm type
  - GetEndpointFromOrchestration: Get the IP:PORT/URI endpoint from an Orchestration Form
  - OrchestrationForm2OrchestrationForm: Helper FB to set the connection to a Orchestration Form type
  - OrchestratorRequestAdp: Request orchestration adapter
  - RequestOrchestrationForm: Request Orchestration Function Block
  - HTTP
    - GetEndpointFromOrchestrationHTTPPartial: Get the endpoint at INDEX from a request orchestration response
    - GetEndpointFromOrchestrationHTTPFull: Get the endpoint at INDEX from a request orchestration response
    - RequestOrchestrationHTTP: Request Orchestration Function Block using HTTP
    - RequestOrchestrationHTTPPartial: Request Orchestration using HTTP with all fields from services to be set
    - RequestOrchestrationHTTPFull: Request Orchestration using HTTP with all fields from services, system and cloud to be set

16. PLC integration IEC61499

PLC integration IEC 61499 module aims at easier connection of industrial PLCs to the Arrowhead framework.

In order to do so, we extended 4diac – an open source PLC framework for industrial automation & control – with function blocks and data types to communicate to the official Arrowhead Framework implementation using HTTP/JSON.
Related documentation and code: https://www.eclipse.org/4diac/

Additionally, we extended some of the Arrowhead core systems (ServiceRegistry, Orchestrator and Authorization) to support OPC UA interface in parallel with HTTP. This eases the connection of the industrial applications to the Arrowhead Framework.

17. Plant Description

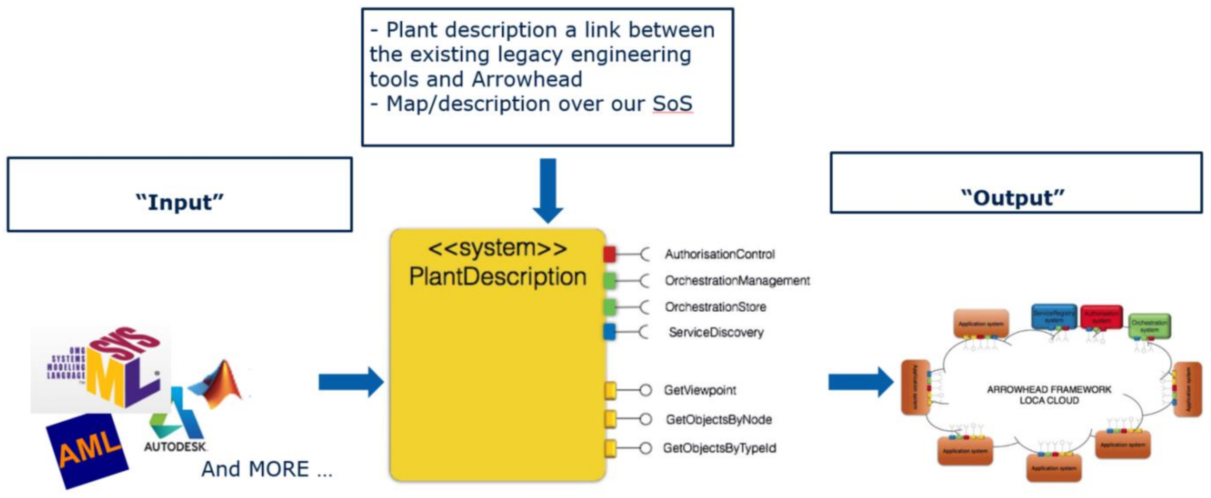
The Plant Description is part of the support core system suite of Arrowhead Framework. The main objective is for the Plant Description system is to:

- Provide a common understanding about the plant
- Provide different actors with different viewpoints to access the dataset from their point of interest
- Provide the Orchestration system with information on which producer a consumer should connect to for a specific service which supports the overall target of the SoS (I.e. provide Orchestration rules)

The Plant Description is a complex system with several different stakeholders (e.g. electricians, network technicians, software developers, control engineers etc.). In order to address the challenge of unifying all stakeholders, the system will be developed in an iterative process where prototypes are proposed and evaluated with the Arrowhead community. At this time, the development is still in the conceptualisation phase focusing on creating the first prototype where four different levels of information will be presented:

- Geographical location, usually part of a hierarchy.
- Properties of hardware (E.g. a sensor's unit, measuring range, precision of measurement etc).
- Function attributes, for example, quality of service parameters such as latency and bandwidth.
- Service attributes, for example: Produced or consumed service/ services (a unique service name), end point of service (e.g. IP address.)

Location	**Product**	**Function**	**Service**
A geographical location, usually part of a hierarchy. Typical attributes: - GPS coordinates - Arbitrary (x,y,z) coordinate system - A descriptive hierarchy.	Physical properties of the product. In the case of a sensor: - Measured unit (C, not K or F) - Measured range (-40 - +80 C) - Precision of measurement (+/- 2 C)	Function attributes, for example, quality of service parameters: - Latency - Bandwidth - If the function is service or legacy based functionality	Service attributes, for example: - Produced or consumed service/ services (a unique service name) - End point of service (e.g. IP address.) - The production or required production pattern (i.e. how often)



18. Python client library

This is a Python library for the creation of Arrowhead Clients.

As the Arrowhead Framework matures, we want 30nt h reach as many developers as possible, 30nt he Arrowhead clients are mostly supported by Java, and a lesser extent C++, the framework won't be readily adopted by developers using other languages. The Python client library will give Python developers a tool to build Arrowhead clients.

Currently the library supports the Service Registry and Orchestration core services, with the plan to have support for Authorization by the end of the year 1.

19. Translation semantics

In an environment where systems are using a wide variety of data formats, data models and semantics, interoperability can be hard to achieve. To enable interoperability in such environments, messages need to be transformed in ways that are hard to predict and expensive to engineer.

The Semantics translator (or message translator) is a system under development that will utilize modern machine learning techniques to translate messages sent from one system to ones that another system can understand, with little engineering effort in between. The plans are that the Semantics translator will be able to utilize a history of messages sent within a system (SoS), semantic metadata and engineering knowledge to create robust translations at run-time.

This system is still being conceptualized in some ways but partners have performed some tests on a very simplified model that show positive results, 30nt he30 not yet near the performance that would be asked of a production environment.

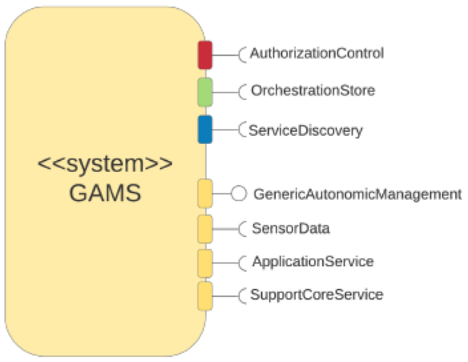
20. FIWARE Interoperability

This service is a plugin to the Translator and can be used to allow Arrowhead-compliant systems to exchange information with an Orion broker using the FIWARE specification. The FIWARE plugin supports bi-directional communication, i.e. an Arrowhead system can push data to FIWARE, or a FIWARE application can exchange information with an Arrowhead system.

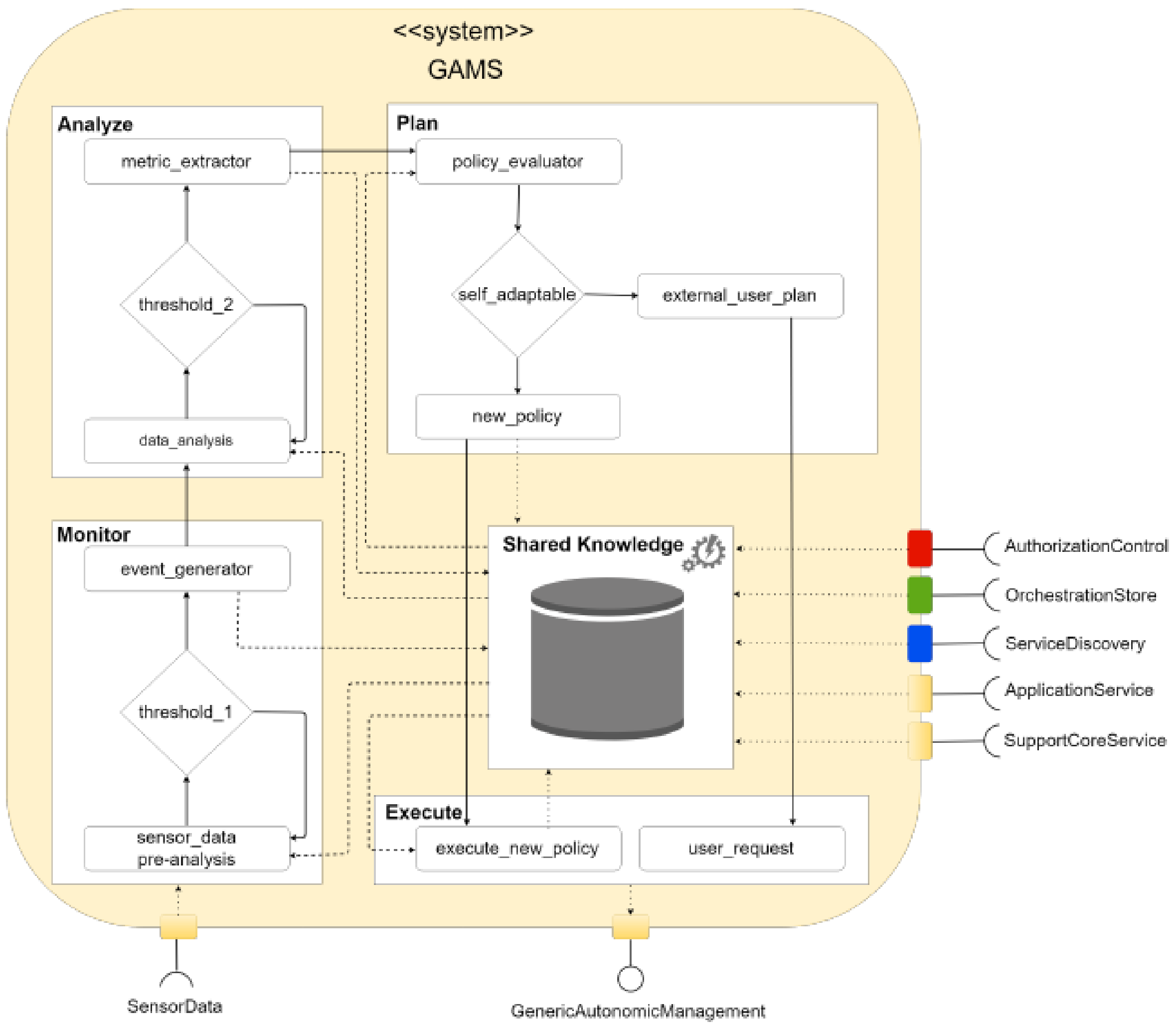
The current development status is that translation between Arrowhead and FIWARE is working for a few types of Arrowhead services (SDD, IDD). Further work is planned to implement support for more Arrowhead application systems.

21. Generic Autonomic Management

The Generic Autonomic Management system (GAMS), shown 31nt he below figure, produces the *GenericAutonomicManagement* service and consumes the three mandatory core services of the Arrowhead framework, *ServiceDiscovery*, *AuthorizationControl* and *OrchestrationStore*. In addition, this system can consume other support core services, such as plant description, workow manager, etc., and application services to build the shared knowledge properly by using semantics and ontologies that already exist 31nt he Arrowhead framework.



The GAMS system, shown 31nt he below, is designed as a component-based REST service (GenericAutonomicManagement service) that can be invoked by different SOA-based frameworks. Additionally, given its generic property, each component of the autonomic control loop has abstract interfaces that can be used by a number of applications systems.



Monitor The Monitor component constantly collects monitoring data from the sensor. The component performs a pre-analysis based 32nt he incoming sensor data and context data stored 32nt he SharedKnowledge. In case there is a signi_cant delta an event is generated. Despite the application system that is using GAMS system, the Monitor abstract interface contains the following functions:

```
monitor [serviceID]
getSensorData (sensorData)
preAnalysis ( )
generateEvent ( )
updateSharedKnowledge ( )
```

Analyze The Analyze component evaluates the events received from the Monitor component with respect to the requirements and context data 32nt he SharedKnowledge. If the requirements cannot be satisfied a change request including a description of the metrics is send to the Plan component. The Analyze abstract interface contains the following functions:

```
analyze [ serviceID ]
getRequired ( required )
getContext ( context )
getEvent ( event )
extractMetric ( )
updateSharedKnowledge ( )
```

Plan The Plan component is able to understand the metrics received from the Analyze component and to derive adaptation policies. The Plan abstract interface contains the following functions:

```
plan [ serviceID ]
getMetric ( metric )
addResource ( )
releaseResource ( )
updateSharedKnowledge ( )
```

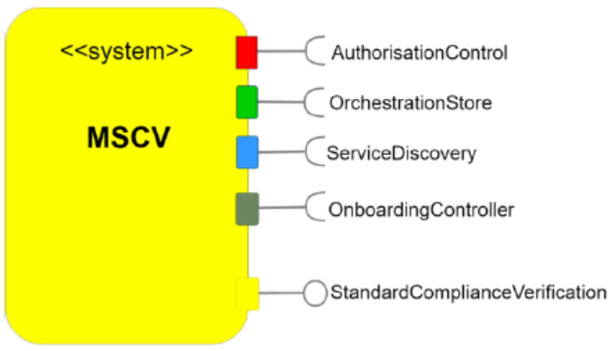
Execute The Execute component receives the policies from the Plan component and executes the derived action via the GenericAutonomicManagement service. The Execute abstract interface contains the following functions:

```
execute [serviceID ]
getPolicy ( policy )
invokeNextAction ( )
updateSharedKnowledge ( )
effectorAdd [serviceID ]
effectorRelease [ serviceID ]
```

22. Monitoring and Standard Compliance Verification

The Monitoring and Standard Compliance Verification (MSCV) system monitors and verifies if the new device/system/service that is interacting with Arrowhead Framework, fulfills the requirements of a specific standard. The standard compliance is defined based on given sets of requirements from which are derived measurable indicator points. Those reflect configurations of systems recommended by standards and guidelines, which help to demonstrate the state of compliance.

The compliance checks will be executed during the onboarding procedure. If the device/system/service is compliant with the standard it can continue the onboarding procedure as described 33nt he OnboardingController SysD. The MSCV system will perform compliance verification based on a set of measurable indicator points, which will be extracted from international standards (e.g., IEC62443-3, ISO27002, etc). The result of standard compliance will decide if the device/system/service can continue with the Onboarding procedure in order to register devices in DeviceRegistry, systems in SystemRegistry and 33nt he33urs33 ServiceRegistry.
The MSCV system is one of the support core systems of the Arrowhead Framework. It provides one service and consumes four services, as shown by the below figure.



The MonitoringStandardComplianceVerification system provides the StandardCompliance-Verification service and consumes the mandatory core services (ServiceDiscovery, OrchestrationStore and AuthorisationControl), and the OnboardingController service. All devices/systems/services should be standard compliant in order to be registered 34nt he DeviceRegistry, SystemRegistry and ServiceRegistry.

23. ESB-CPN

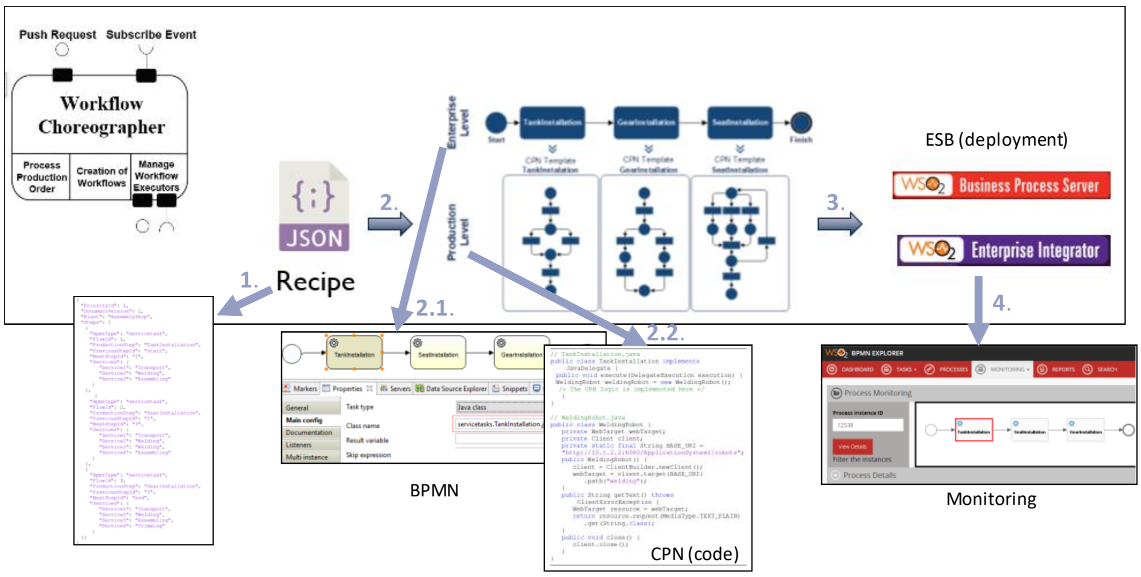
The aim of integrating the Enterprise Service Bus with a Clored Petri Nets-based engine is to provide a full coverage of process control from the business to the manufacturing floor level. The concept utilizes the Arrowhead Framework, and its Event handler and Workflow Choreographer Systems beside the mandatory core systems and services.

The systems and services provided by this implementation are a variant or alternative of the Workflow Manager (one of the Supporting Systems in the Arrowhead framework). In this alternative, the Workflow Manager is a data driven approach with a workflow manager (Workflow Choreographer) and executor entities (Workflow Executors). A homogeneous integration concept of how the framework can support processes at business-, production floor- and workstation levels is proposed. The solution integrates Business Process Modelling Notation (BPMN) and Coloured Petri Nets (CPN)-based task execution using an Enterprise Service Bus (ESB) infrastructure for process deployment, execution and monitoring.

The Workflow Choreographer is the main engine of automated production. From the production point of view, it processes the production orders and implements the production recipes. The Workflow Choreographer creates and executes workflows based on predefined templates according to the specific procedures. The production recipe uses these templates based on production steps, and the Workflow Choreographer creates the workstation specific Workflow Executors (WE), which are the real production accomplishments of the workstations. It executes the related activities and tasks according to the production recipe, which are given to workstations for performing modifications on goods. Based on the production recipe, the devices and systems in a workstation will be coordinated to complement the related workflow step.

The figure shows the process steps:

1. Produces Production Recipes from Production Orders
2. Creates workflows at Enterprise Level (BPMN) and Production Level (CPN)
3. Manage workflows with Workflow Executors (deployment of BPMN and CPN models)

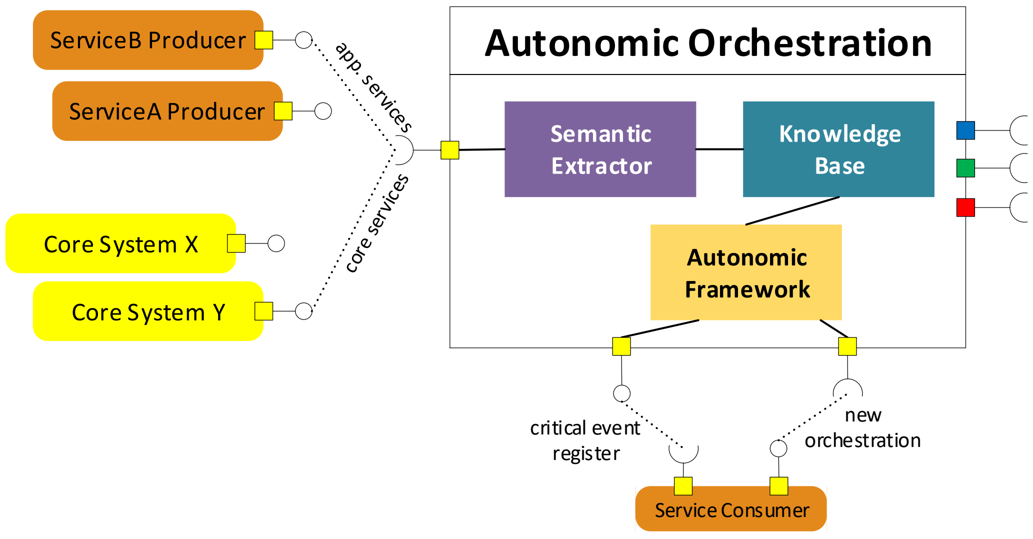


The ESB-CPN proposal is now in the implementation stage (implementation ongoing). It is not mature enough to be considered a released version. Works to accommodate the Workflow Choreographer so it can be entirely manage by the Arrowhead framework are necessary. A simpler Workflow Description Language is also necessary and the team is working on this. Finally, automation on the conversion of CPN into Java is required.

24. Autonomic Orchestration System

The Autonomic Orchestration enhances service orchestration with self-adaptation capabilities. Specifically, the Autonomic Orchestration system enables application systems to register their policies of service adaptation with semantic rules. For instance, the application systems could use semantic rules to specify abnormal 35nt he35urs of their consuming services and how they would react whenever such anomaly detected by the Autonomic Orchestration system (e.g. changing to similar services). The Figure below illustrates the core components within this core system as well as its interaction with other Arrowhead systems within the local cloud.

The Autonomic Orchestration system consumes the three core mandatory systems in order to construct the information models of the services within the cloud. In addition, this system also consumes other core systems such as System or Device Registry and Plant Description to build a complete knowledge base about the SoS architecture. The system also consumes other application services for monitoring their abnormal 35nt he35urs specified by the semantic rules



At this moment, the system is under conceptual design. A prototype related to a smart factory usecase is under development now and will be demonstrated at the general Arrowhead Tools Workshop in November.

25.Exchange Negotiation Service

The Exchange Negotiation Service allows distinct stakeholders to negotiate about and commit to contractual rights and obligations. The service itself provides the primitives and protocols required to perform such negotiations and prove what changes in rights and obligations have been committed to, while any system actually producing the service must define the logic and contractual terms required for any changes to rights and obligations to have legal bearing. The service is intended to be useful for negotiating everything from payment terms to frame agreements, and aims to become an integral component for buying or selling access to Arrowhead services or to concrete data objects, such as digital twins.

Work 36nt he third iteration of the service has recently begun. SD and IDD have been submitted. A conference paper about the service has been submitted, a Java-based implementation is ongoing.

26. Safety Manager

The main objective of the Safety service is to supervise the status of different Arrowhead compliant services that are registered in a local cloud. Based on defined contracts, it will ensure the reliable and safe operation of these services.
Its role can be defined as unsafe scenarios detector. When it detects an unsafe scenario, it starts a safe process to protect the behaviour of the system.
The approach or concept developed has the following characteristics:

- Arrowhead Local cloud has a Safety related service.
- Different services are monitored by the Safety Monitor.

These services are based 37nt he RESCO software components. they have to be offered as Arrowhead compliant services and they provide information about the status of the controlled system in model terms at runtime.

- The monitored information is checked by the Safety Manager. For doing that, safety rules or properties common language is defined and the safety rules will be defined using this common language in each use case.
- When an unsafe scenario is detected, a safe process starts and the services involved 37nt he use case are updated to a graceful mode.

Status: the conceptualization phase finished and first version in an academic use case developed. Next steps: develop in a simple industrial use case.

27. Arrowhead-compatible C/C++ clients

UTIA provides support for Arrowhead compatible C/C++ Clients: Provider (server application) and Consumer (client application) for Xilinx Zynq Ultrascale+ 64 bit devices.

The Xilinx Zynq Ultrascale+ devices contain 4x A53 Arm processor (64bit), 2x R5 Arm real time processor (32bit) and programmable logic area serving for HW accelerators.

The ARM part of the device is running the Debian “Stretch” OS, supports C/C++ compiler. The device is placed on an industrial PCB module together with 2 GB of DDR4 memory.

The started development is motivated by the needs of the ARCELIK use case – a partner of UTIA. Automation of fast data measurement of transient MOSFET power transistors switching in Power supply units produced by the company ARCELIK is needed. Automation of fast switching transient events is needed for each produced power supply unit for different combinations of input AC power voltage and different output loads. The results of automated measurements will be processed by the Arowhead framework. The A/D transient event measurements require HW system capable to perform A/D data measurement with 1G samples/s rate. UTIA and EDI collaborate with company ARCELIK in the ARCELIK use case. UTIA and EDI developers have visited the ARCELIK plant in Istanbul and the required system was jointly specified there.

In WP3, the main objective of the already started UTIA design/development work is to provide:

- Debian board support package for family of Zynq Ultrascale+ devic based industrial modules on carrier PCB hosting one FMC data acquisition card.
- Low level SW support (drivers) for family of data acquisition cards (FMC HPC) capable to perform 1Gsample/sec A/D and D/A data measurement tasks needed in the ARCELIK use case.
- Arrowhead compatible SW C/C++ Clients for Debian OS on Zynq Ultrascale+. Provider (server application) and Consumer (client application) running as C/C++ user applications Xilinx Zynq Ultrascale+

UTIA have started development of board support packages (BSPs) for family of specified industrial modules with different size of programmable logic and different predefined functions.

UTIA have also started development of HW design capable to perform A/D data measurement with 1G samples/s rate. Work on board support package generation script is progressing well. It is based on tcl script. It is compatible with Win7/Win10 or Linux Ubuntu 16.04 LTE and requires Xilinx Vivado 2018.2 HW development tool chain and the Xilinx Petalinux 2018.2 kernel configuration scripts.

UTIA is currently capable to perform on Zynq Ultrascale+ the single channel for A/D data and D/A measurement with 1G samples/s rate with the Analog devices data acquisition board AD-FMCDAQ2-EBZ.

The specified Zynq Ultrascale+ system supports the Arrowhead framework compatible SW C/C++ Client code templates. Currently only non secure 1Gbit Ethernet communication in local cloud is supported. The arrowhead services are running on RaspBerryPi 3B board or on Ubuntu PC.

28. Conclusions

The core concepts of the Arrowhead Framework keeps expanding and maturing. The proceedings of its conceptual growth, its design and implementation are summarized within this document. There are two kinds of growth regarding the framework.

First, it is the natural maturation of core systems and services – through reviews, re-design steps, implementational fine-tuning, and then again reviews.

Second, the framework keeps refining the already included concepts, and keeps adopting new ones that are necessary to cover the interoperability, integrability and engineering needs of real-life scenarios.

29. Revision history

29.1 Contributing and reviewing partners

Contributions	Reviews	Participants	Representing partner
Arrowhead Framework Mandatory Core v4.1.3	LTU, Bosch, Eurotech, evopro, BME, IQL, ISEP, IFAK, UTIA, MGEP, MON, Jotne, VTT, Wapice, HIOF, Polito	Szvetlin Tanyi,Tamas Bordi, Gabor Majoros, Rajmund Bocsi	AITIA
Service Registry, Authorization, Orchestrator, Gatekeeper, Gateway	LTU, Bosch, Eurotech, Evopro, BME, IQL, ISEP, UTIA, MGEP, MON, Jotne, VTT, Wapice, HIOF, Polito	Szvetlin Tanyi	AITIA
Data Manager, FIWARE Interoperability	BME	Jens Eliasson	LTU
Extended Historian	BME, LTU	Mario Thorn	IFAK
Translator, Translation Semantics	BME	Jens Eliasson, Jacob Nilsson	LTU
Device Registry, System Registry, Onboarding Controller, Generic Autonomic Management, MSCV	evopro, AITIA	Ani Bicaku, Silia Maksuti, Markus Tauber	FHB, LTU
4DIAC, PLC integration IEC61499	BME	Jose Cabral	Eclipse
Plant Description	BME	Jaime Garcia, Niklas Karvonen	LTU
Python Client Clibrary	BME	Jacob Nilsson	LTU
ESB-CPN	AITIA, LTU	Felix Larrinaga, Daniel Kozma, Pal Varga	MGEP, BME
Autonomic Orchestration System	HIOF	An Ngoc Lam	HIOF
Exchange Negotiation Service	BME	Emanuel Palm	LTU
Safety Manager	BME, LTU	Miren Illarramendi, Daniela Cancila	MGEP
Arrowhead-compatible C/C++ clients	BME, AITIA	Jiri Cadlec	UTIA
Intro, Summary, Document integrity	LTU	Pal Varga	BME

29.2 Amendments

No.	Date	Version	Subject of Amendments	Author
1	2019.10.01.	0.1	Initial Version	Pal Varga
2	2019.10.20.	0.9	Partner contributions added	Pal Varga
3	2019.10.25.	0.97	Further contributios added, release for review	Pal Varga
4	2019.11.08	1.0	Review comments integrated, and other modifications	Pal Varga

29.3 Quality assurance

No	Date	Version	Approved by
1	2019.10.04.	0.1	Jerker Delsing
2	2019.11.07	0.97	Jerker Delsing
3	2019.11.08	1.0	Jerker Delsing