Document title: Arrowhead Tools Deliverable D3.3

Version	**Status**	**Date**
1.0	Final	2020-11-22

Authors: Johannes Kristan, Janina Schuster, Pal Varga

Deliverable D3.3

“Consolidated SOA framework platform architecture, design and sw implementations of core systems – Y2”

Editors:	Johannes Kristan, Janina Schuster Pal Varga (johannes.kristan@bosch.io - janina.schuster@bosch.io – pvarga@tmit.bme.hu)
Work package leaders:	Pal Varga, Szvetlin Tanyi Daniela Cancila (pvarga@tmit.bme.hu - szvetlin@aitia.ai - daniela.cancila@cea.fr)

Abstract

This document constitutes deliverable D3.3 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 at its version 4.2. Furthermore, this document presents the plans and status for year 2 of the Arrowhead Tools project, regarding the core components of the Arrowhead Framework, and very importantly, its integration with various Eclipse IoT project parts.

This deliverable is Released by Work Package 3 (WP3) of the Arrowhead Tools project, which is responsible for the consolidated Service Oriented Architecture. This includes the core concepts of the Arrowhead Framework, their design, integration, and the Reference Implementation as well.

ECSEL EU project 826452 - Arrowhead Tools
Project Coordinator: Professor Jerker Delsing | Luleå University of Technology

ARROWHEAD TOOLS

Table of Contents

1 Introduction

The Work Package 3 in the Arrowhead Tools project is responsible for the consolidated Service Oriented Architecture. This includes the core concepts of the Arrowhead Framework, their design, integration, and the Reference Implementation as well.

The Arrowhead Tools project introduced two very important additions into the life of the Arrowhead Framework: one is the need for long-term governance; another is the interaction and integration with external frameworks and tools. Long-term governance is now driven by the fact that Arrowhead IoT became officially part of the Eclipse project family. Contribution to source code, release and issue tracking are all very well regulated in Eclipse, which means that the implementation-related issues are on very good track (and hard to get drop off that track actually). The other important addition is the interaction and integration with other frameworks and tools. A great part of this has also accomplished by the official agreement between Eclipse Foundation and Arrowhead Tools: now various Eclipse projects are open to actively collaborate with Arrowhead, since they bear the same license.

The Chapters of this deliverable can be categorized into three sets. The first is the description of Arrowhead Core Systems at v4.2 of the reference implementation. The second set is those chapters that deal with External Tool interactions and integrations (such as Eclipse Vorto, Ditto, Hono, Hawkbit, or Kapua and Kura, among others). The third set are those concepts and implementations that have been around or newly popped up, based on industrial requirements, within the Arrowhead Tools project or its predecessors, and have been created or further improved in the previous period – and continue to do so in the upcoming project periods.

2 Arrowhead Framework Mandatory Core – v4.2

The Arrowhead Framework core went from v4.1.3 to v4.2 in this period which is backward compatible with the previous version. Beside correcting reported issues, various additional changes were designed and implemented. Additional to the already released core systems:

- Service Registry,
- Orchestrator,
- Authorization,
- Event Handler,
- Gatekeeper,
- Gateway,

in this period, further supporting core systems have been released officially, namely:

- Certificate Authority,
- QoS Monitor,
- Onboarding Controller,
- Device Registry,
- System Registry,
- Workflow Choreographer.

The complete documentation of the v4.2 reference implementation - with user guides, developer guides and core system API details - is publicly available at: **https://github.com/arrowhead-f/core-java-spring**

While the the brief description of all core systems were included in D3.2, please see a brief ”motto” of those supporting core systems that have been officially released:

- The main purpose of the *Certificate Authority* supporting core system is issuing signed certificates to be used in the local cloud. The issued certificates may be revoked from the Management Interface. Systems may check whether a certificate has been revoked, and refuse their acceptance.
- The purpose of *QoS Monitor* supporting core system is providing QoS (Quality of Service) measurements to the QoS Manager (which has became part of the Orchestrator core system for implementation reasons).
- The purpose of the *Onboarding Controller* supporting core system is to be the entry board for the onboarding procedure. The onboarding controller sits at the edge of the Arrowhead local cloud. It is not only reachable from within the cloud by authorized systems, but also from the public through its "accept all" interfaces. Any client may authenticate itself through an Arrowhead certificate, through an authorized manufacturer certificate, or simply through a shared secret.
- The *Device Registry* supporting core system provides the database, which stores information related to the Devices within the Local Cloud. The purpose of this System is therefore to allow:
    - Devices to register themselves, 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.
    - Generate a client certificate which can be used by the Device to register its Systems
- The *System Registry* supporting core system provides the database, which stores information related to the System of the currently actively offered Services within the Local Cloud. The purpose of this System is therefore to allow:
    - Devices to register which Systems 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.
    - Generate a client certificate which can be used by the System to offer Services
- The *Workflow Choreographer* supporting core system makes it possible to execute pre-defined workflows through orchestration and service consumption. Each workflow can be divided into three segments, namely *Plans*, *Actions* and *Steps*. *Plans* define the whole workflow by name and they contain *Actions* which group coherent *Steps* together for greater transparency and enabling sequentialization of these *Step* groups.

3 Eclipse IoT Integration

The Eclipse Foundation was created in 2004 as an independent not-for-profit organization, with the aim to foster the development of open source projects and provide a vendor-neutral community where individuals, as well as corporations can collaborate. It acts as the managing entity for the Eclipse community, which consists of individual developers and corporations. The foundation is governed by an independent board of directors and is host to multiple working groups. One of them is the Eclipse IoT working group, which is actively developing open source technology for the Internet of Things. To this point, the working group combines the work on over 30 different projects that each target another aspect of IoT deployments ranging from implementations for actual devices and gateways over protocol implementation to components for building IoT cloud backends.

One of the organizational achievements – supporting long term governance – in this period was that the Eclipse Arrowhead project initiative has been accepted by the Eclipse Foundation.

This included various organizational and administration steps, although technically the most challenging was the code review that the Arrowhead Core Systems had to go under. This was successful, so the main obstacles of integrating Eclipse IoT projects and the Eclipse Arrowhead IoT project have been removed. The following sections detail the various Eclipse IoT projects that have commonalities – and from now on, working towards similar goals partially – with the Eclipse Arrowhead IoT project.

3.1 Rationale for Integration

The accumulation of IoT technology in the Eclipse IoT Working Group is one of the largest collections in the field, with many active corporate contributors from the industry, including a series of production-ready software artifacts. Each project proposed under the umbrella of this working group undergoes a standardized review process, guaranteeing certain quality measures for the development process like the reporting of vulnerabilities. Furthermore, the Eclipse foundation defines an intellectual property due diligence process, ensuring a project does not infringe anyone's intellectual property rights, e.g. by license clearing all software dependencies. Moreover, some of the projects, even though developed as standalone-applications, are designed to work together to build feature rich, reliable IoT-Applications and scenarios. The working group provides software that ranges from cloud platforms to applications that run on constrained devices and is therefore very eclectic. All those reasons make Eclipse IoT technology a good complement to the Arrowhead Framework, which allows to achieve the targets of the Arrowhead Tools project.

3.2 Technical Details to Eclipse IoT Integration Evaluation and Results

The Eclipse IoT working group differentiates between three hierarchical layers within the IoT infrastructure [1] as depicted in

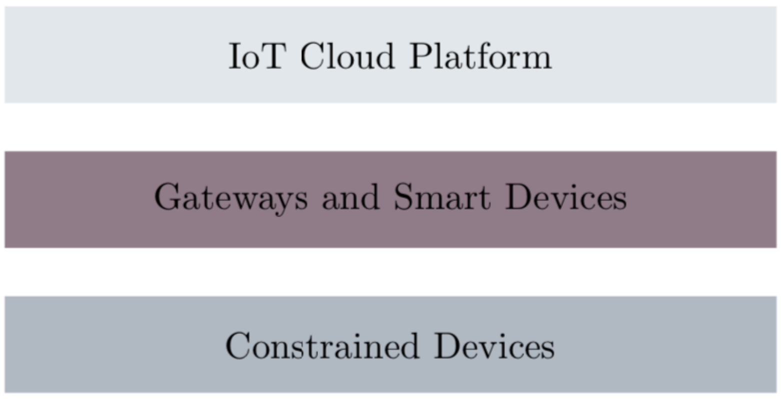


Figure 3.1. Constrained sensors and actuators at the bottom that measure data and execute commands but do not process or evaluate information due to low compute power. These devices interact with gateways at the edge, which build an intermediate layer to the IoT cloud platforms. The cloud enables high scalability and fast processing of large data volumes, data aggregation and long-term analysis. Eclipse IoT currently comprises approximately 35 different open source projects [2] from different fields of application. Most of the projects can be associated with one of the three layers.

We evaluated to which extent these existing technologies do harmonize with the service-oriented Arrowhead concept in an industrial IoT context. As Eclipse IoT is not primarily aimed at industrial use cases, not all projects are suited. However, the Eclipse IoT working group did identify their open source projects that are ready for potential Industry 4.0 application in a white paper [3]. Table 1 lists all projects mentioned in that whitepaper as suitable and a classification regarding their position in the Eclipse IoT working group stacks, if possible.

The less hierarchical nature of the Arrowhead Framework does potentially interfere with the 3-tiered approach of the Eclipse IoT working group. At least the projects belonging to the "IoT Cloud Platform" layer generally follow a more centralized principle, whereas the Arrowhead Framework advocates a distributed approach. On the other hand, the SOA-structure of Arrowhead allows simple replacement and integration of new modules and is hence ideally suited for incorporating existing software components. Moreover, the Eclipse IoT Cloud platform projects feature a mostly homogeneous software stack, relying mainly on Java, Spring [5], Vert.x [4] and other open source modules.

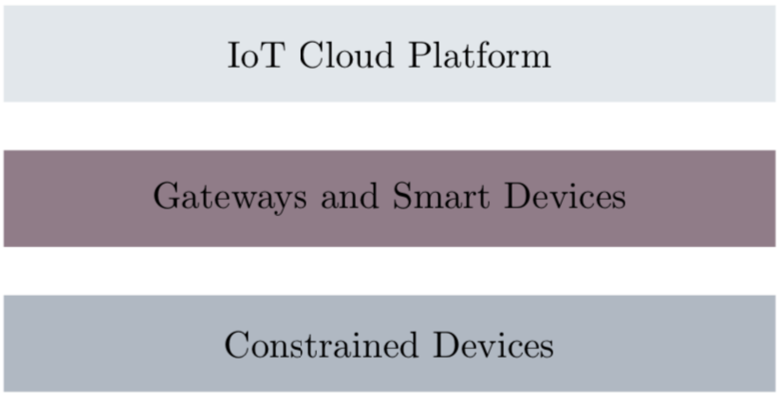


Figure 3.1 Eclipse IoT Stack

Table 3.1 Eclipse IoT Projects with Relation to Industrial Use Cases

Project	**Category**	**Layer**
Eclipse Milo	Data Aggregation	
Eclipse Mosquitto	Data Aggregation	Cloud
Eclipse Paho	Data Aggregation	Devices
Eclipse Unide	Data Aggregation	
Eclipse OM2M	Data Aggregation	
Eclipse 4diac	Data Aggregation	
Eclipse Kura	Data Aggregation, Security, Digital Twin	Gateways
Eclipse Leshan	Security, Device Management	Gateways, Cloud
Eclipse Keti	Security	
Eclipse Wakaama	Device Management	Devices
Eclipse Kapua	Device Management, Event Management and Data Analysis	Cloud
Eclipse hawkBit	Device Management	Cloud
Eclipse Hono	Event Management and Data Analysis	Cloud
Eclipse Ditto	Digital Twin	Cloud

3.2.1 Applications within the Arrowhead Framework

The functional, intentional and technical fit of the various Projects of the Eclipse IoT Working Group for their integration with the Arrowhead Framework was studied in [31]. It provides a short introduction and evaluation of each eligible Eclipse IoT project and its usability within one or multiple Arrowhead systems. One result is that almost none of the projects can be used without adaptation or without at least combining them with other modules. However, many of them could potentially build the foundation of an Arrowhead component or at least function as a structural archetype. The following enumeration only considers Eclipse components, which have a connection to at least one of the Arrowhead systems. Figure 3.2 gives an overview of the considered Eclipse IoT projects and their possible connection with respective Arrowhead services.

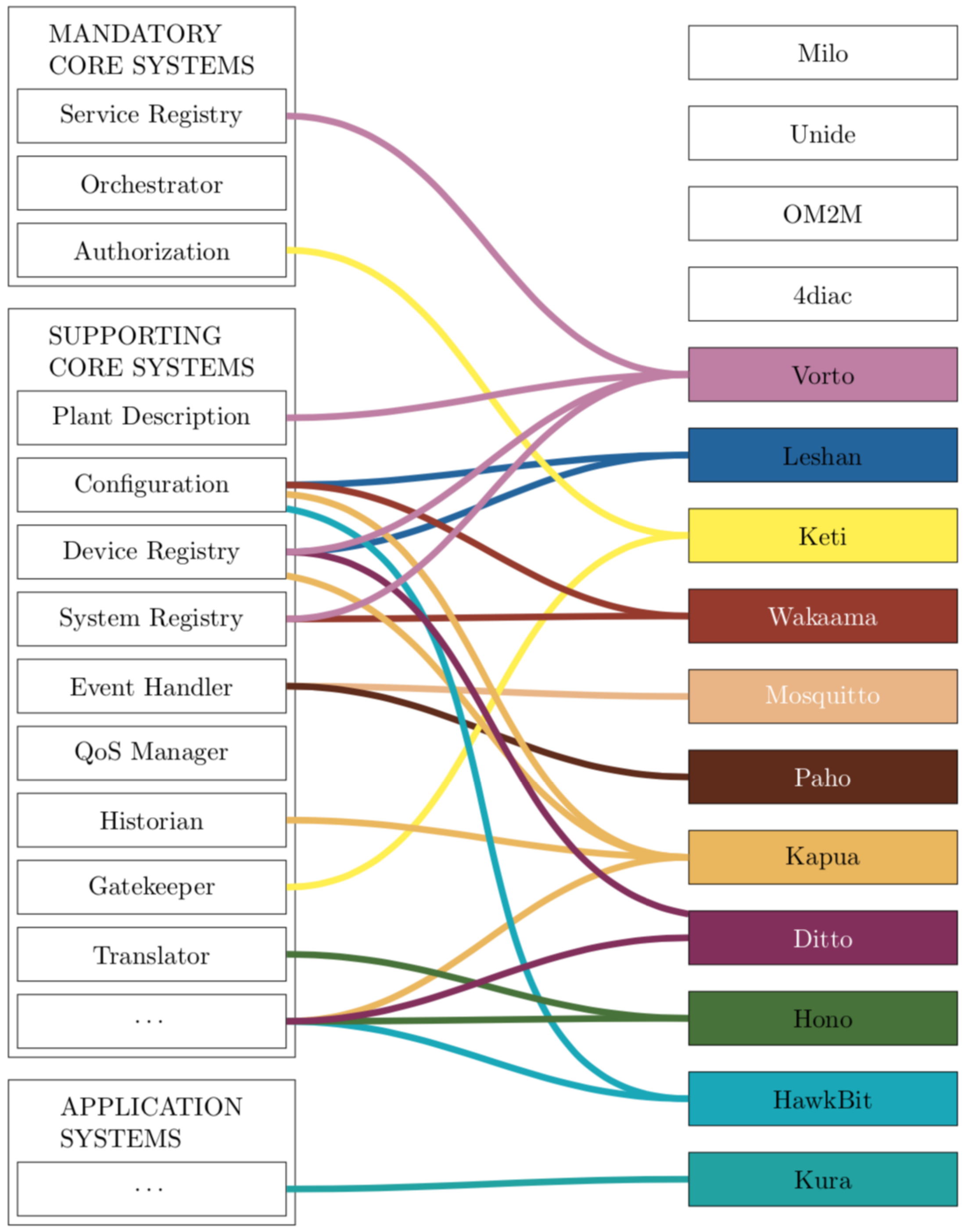


Figure 3.2 Comparison of Arrowhead systems (left) and eligible Eclipse IoT components (right). A connection indicates a potential application of an Eclipse IoT project in one of the generic Arrowhead systems. The order of Eclipse IoT systems is arbitrary, as well

Based on those results Bosch.IO focuses on the integration of the components of the projects Eclipse Hono, Eclipse Ditto and Eclipse hawkBit. Eurotech is focusing on the integration of the projects Eclipse Kura and Kapua (see Chapter 4). Therefore, in the next sections the projects Bosch.IO is focusing on are introduced here and Kura and Kapua in the next chapter. Orthogonal to the Eclipse IoT projects named before, which support the setup of Systems of

Systems, is Eclipse Vorto. It supports the modelling and integration of devices with the aforementioned projects. Eclipse Vorto, even though it is discussed in other Work Packages (4, 5, 6), has relations to the work in Work Package 3 and therefore is shortly introduced here.

3.2.1.1 Eclipse Vorto

Eclipse Vorto is a Domain Specific Language (DSL) for digital twins within the IoT world, inspired by Java [6]. It offers a generic possibility to describe the characteristics of a hardware device, such as name or size and attributes like temperature or location, but also its functionalities. Vorto also enables developers to define dependencies between different devices and provides a repository to share generic building blocks to reuse for new device definitions. Moreover, Vorto allows the creation of code generators, enabling the generation of device integration stubs.
Some of that functionality could be used for building a device-, system-, or service- registry. The standardized description could easily be used to create a device-registry API, which offers detailed information about each deployed device. More information on the Arrowhead definition of the service, system, and device registry can be found in [8] and [7]. The reusability aspect of Vorto and its building blocks repository allows the simple creation of new device descriptors based on existing ones. This approach of consecutive modules would also facilitate the implementation of more complex structures, such as entire systems or services. Hence, a system- or service- registry, depending on the use case, is conceivable as well. However, with rising complexity, Vorto's DSL might not be the best approach.
Speaking of complex systems, a plant description model could also be designed using an Eclipse Vorto definition. The Arrowhead plant description system provides a basic overview about a plant's layout (for more information see [9]). As Vorto also offers a meta- model to describe the relationship between different modules and their dependencies, Vorto could be an alternative to non-standardized modelling techniques. Moreover, using the same DSL for multiple systems in the same Arrowhead implementation presumably saves resources.

3.2.1.2 Eclipse Ditto

Ditto provides a service that manages so called "digital twins" [18]. In Ditto, a digital twin is a software pattern where all relevant subjects from the physical world in an IoT context, such as sensors or machines but also products during their manufacturing cycle, are each represented in the digital world. Each representation has its own attributes and characteristics, such as IDs, names or sensor values, and acts as a single point of truth for each given device. Ditto organizes the access to these digital twins, providing a safe and multi-protocol-capable API that allows integration with other backend infrastructure and features individual access policies.
Within the Arrowhead context, Ditto could be used for implementing a device registry. However, this device registry would not just be a repository for all existing devices, as the Arrowhead Framework declares it (for more see [7]), but even more handle changing device states.
Hence, it could be argued that such an implementation in fact exceeds the scope of a device registry in the Arrowhead context. Therefore, it seems more sensible to introduce a new digital twin core system. This system would not just be a registry for all devices but would also persist its current state and makes it available at any time. Moreover, Ditto seamlessly integrates with Eclipse Hono, making it a perfect match in serving even more different protocols and devices.

3.2.1.3 Eclipse Hono

Hono is a multi-protocol IoT hub, connecting large numbers of devices with a cloud- based business application [21]. Its main benefit is its diverse protocol compatibility among device interfaces, such as HTTP, MQTT, CoAP, and AMQP 1.0. However, business applications have to connect using AMQP 1.0. Hono supports three kinds of data exchange. First, it allows the transmission of telemetry data, such as temperature or humidity information from sensors to a business application. Second, Hono also allows the transmission of events, indicating e.g. the completion of a process step in an industrial use case. Third, business applications can initiate so called command and control interaction patterns with selected devices to trigger certain actions, such as adjusting a heating system. There are two imaginable applications of Hono within the Arrowhead context.

First, Hono does already cover an important aspect of every IoT infrastructure, which is linkage of hardware devices with business applications on higher levels in the IoT hierarchy. Especially, devices that do not expose RESTful APIs via HTTP and potentially rely on a broker infrastructure, such as used by publish-subscribe protocols (e.g. MQTT) benefit from a centralized device hub. Moreover, Hono provides standardized communication patterns as described above.

In addition, Hono's multi-protocol capability could be repurposed to build a translation system. The purpose of the Arrowhead translation system is to provide a bridge betwen systems that do use the same protocol for communication (for more information see [19]). This translation system would enable easy transformation of e.g. MQTT to HTTP or CoAP.

3.2.1.4 Eclipse HawkBit

HawkBit is another device management solution targeted at software rollouts [20]. The application comprises the core module — the update server — a management UI and multiple APIs. The first API exposes a REST-based management interface for third-party applications, the second API provides direct device integration via REST and polling, and the third API offers device integration via AMQP. This latter endpoint enables the incorporation of intermediate applications to feature different protocols, such as OMA-DM, LWM2M, or proprietary ones. HawkBits' software rollout process allows sophisticated configuration options, such as grouping of devices, cascading deployment, and fine-grained monitoring.

Hence, Eclipse HawkBit appears to be a potential candidate for an Arrowhead configuration system implementation. Especially its extensibility for further protocols allows the integration of a wide variety of devices, but also enables the combination with Eclipse Leshan to feature LWM2M-managed devices. However, configuration and update procedures are highly critical, as they could potentially compromise a multitude of devices by implanting malicious code. Therefore, the existing HawkBit authorization measures would need to be integrated with the Arrowhead authorization system.

3.2.2 Summary

As the evaluation of the existing Eclipse IoT stack reveals, a lot of different frameworks and tools are available that can either complement the existing Arrowhead Framework or could serve as industry grade implementations of some of its core services.

As a result of the evaluation and the identified technical needs of some of the use cases (especially in Work Package 9). Bosch.IO is concentrating on the integration of the Eclipse Projects Eclipse Hono, Eclipse Ditto and Eclipse hawkBit.

3.3 Integration in Arrowhead Framework

As described in the previous section, the Eclipse IoT working group provides a wide variety of projects that could be integrated into the Arrowhead concept. As stated earlier Bosch.IO focuses on the integration of

- Eclipse Hono as a device hub
- Eclipse Hono as a translation system
- Eclipse hawkBit as a configuration system
- Eclipse Ditto as a digital twin system

All three of the mentioned Eclipse projects are sophisticated, production ready IoT applications and were selected for further analysis, as they were identified to be best equipped for an Arrowhead integration.
To fully comprehend the following explanations of the developed proofs of concept, the chapter begins with a short introduction into various technical concepts and frameworks that are used for the implementation.

3.3.1 Relevant Technology Involved

To understand the descriptions and implementations of the practical integration attempts, the following section introduces the technology used. The description of the Arrowhead Framework is already done elsewhere. However, in this section we discuss the Arrowhead token-based authentication mechanism, which is needed in every integration attempt. The section also provides a short introduction to reactive programming, the development paradigm which is used to realize the integrations. This includes the presentation of two reactive frameworks, namely Akka and Eclipse Vert.x. And finally, a short description of the involved messaging protocols and their conceptual differences is given.

3.3.1.1 Token based authentication

As explained above, the current Arrowhead reference implementation allows other systems to authorize, using a token based approach, called JSON Web Tokens (JWTs) [22]. JWTs enable an independent authentication authority, such as the Arrowhead authorization system, to issue a security token, containing additional information about the client, called claims. Generally, there exist two different types of JWTs, which extend the specification: JSON Web Signatures (JWS) [24], and JSON Web Encryptions (JWEs) [23]. The former guarantees that a service consumer is who it claims to be and ensures that the payload is unaltered. This is done by using the issuer's public key to check the attached signature. However, the payload, which might contain sensitive information, is not encrypted in this case. JWEs, as the name suggests, provide payload encryption by using the receiver's public key and hence protect the payload from being visible to anyone else than the receiver.
JWS consist of three parts — header, payload and signature — separated by dots. The header contains information about the signing/encryption algorithm used, the payload encapsulates the claims in JSON notation, and the signature is used for validation.
A JWE consists of 5 parts, a header, an encrypted symmetric key, an initialization vector, the actual ciphertext, and an authentication tag. The symmetric key is used for content encryption/decryption and is encrypted asymmetrical. The initialization vector and the authentication tag are needed for encryption and validation.
The JWT standard also allows the combined usage of both versions, resulting in a nested JWT, where a JWS is embedded as the payload of a JWE. This method is used by the current

Arrowhead authorization system implementation, which is necessary due to the fact that not all participating systems might use transport based encryption. However, the token claims contain information that equips a potential attacker with information about the local cloud architecture. Additionally, the signature contained by the encompassing JWS, enables the validation of the identity of the service consumers.
The Arrowhead reference implementation uses the combined JWT approach (JWS inside JWE) in a custom authorization flow. This implies e.g. that the encoded claims do not follow any standardized naming convention. However, there exists a more sophisticated and complex token based authentication mechanism, called OpenID Connect [25], which is based on JWTs for content encoding and the OAuth [26] authorization flow. Eclipse Ditto, like many other open source components, provides authentication and authorization via OpenID Connect, which is not compatible with the Arrowhead mechanism. This is due to the fact, that OAuth and OpenID connect are primarily employed in use cases involving HTTP and an active user entering his or her credentials into a UI. However, RFC 8705 [27] proposes an authorization flow, based on client certificate authentication, which would then enable more automated scenarios. Some OpenID connect authorization servers, such as Keycloak [28] already provide client certificate-based authentication.

3.3.1.2 Reactive Programming in Vert.X and Akka

The Arrowhead Framework does not specify the number of deployed devices, which is bound to the respective usage scenario. However, especially in IoT scenarios, the number of interconnected devices being part of the system might not only be generally high, but also vary over time, as the system evolves. Hence, the Arrowhead Framework has strong requirements regarding scalability, i.e. an Arrowhead-based system and all its comprising components should be able to adapt to changing parameters. To fulfill the required scalability requirements, our integration relies on the reactive programming approach, introduced in the following.
When designing new Arrowhead-ready components for the Internet of Things it is important to examine the usual application scenario first. A potentially large number of devices, most of which being limited in their resources, meaning processing power and storage capacity, exchange messages over a network. The more complex and thus better equipped components are responsible to bring in a certain degree of reliability to this system, which is especially necessary when it comes to highly critical scenarios, where a lost message can result in high costs. Hence, it is important to guarantee message delivery where necessary by employing Quality of Service (QoS) measures, which can be achieved by using a message broker infrastructure as explained in the next section. Moreover, the more complex application components need to be able to handle large amounts of requests, which can lead to overloaded servers.
To achieve a reliable infrastructure, which is capable of handling many requests within reasonable time, and which is resilient, it is important to select the right technology stack. The Reactive Manifesto [Boner2014] advocates a development approach, focusing on four key aspects, namely responsiveness, resilience, elasticity, and being message driven. These specifications are defined as follows [30]:
Responsive: Reactive systems should always be able to respond to new requests rapidly and provide consistent quality of service
Resilient: Occuring failures must be contained within the originating component of the reactive system and an error may not lead to a crash of the entire system.

Elastic: To guarantee responsiveness, even under high workloads, the system needs to be able to adapt, by dynamically increasing allocated resources.
Message driven: To achieve the above-mentioned goals, a reactive system consists of several microservices, interacting by transmitting messages. This guarantees the high degree of decoupling and isolation, which allows for scalability, resulting in resilience and elasticity.
Obviously, these characteristics are favorable in almost any software application, however in Industrial IoT use cases they are more necessary than ever. Imagine a smart home scenario, where users are able to control their light bulbs with their phone. In this example, failures are by far less critical, than in a complex manufacturing line, where an error can result in high financial losses.
In the conventional imperative programming paradigm, programs make direct use of OS threads to utilize the available resources — meaning processing power — as efficiently as possible. Transferring this model to server applications, each incoming request will be delegated to a forked OS thread, which, for instance, handles a blocking database request in the background and subsequently returns a response to the client. In this programming model, every instruction gets executed sequentially and blocking I/O operations, causing a system call, result in a thread context switch. However, scaling this approach eventually fails, as thread management, including context switching, comes with a high overhead of workload with numerous concurrent requests. Even the usage of thread pools ultimately limits the number of active connections.
The reactive programming approach is providing a different concurrency model, opposing the thread-per-request strategy. Instead, reactive systems handle requests in an asynchronous, non-blocking way, where work-intensive tasks or I/O operations do not block an entire OS thread.
Currently, there exist a few frameworks for developing reactive systems, two of which are presented here, namely Eclipse Vert.x and Akka. Both technologies provide a toolbox for developing responsive microservices. They are also the basis of two of the projects planned to be integrated with the Arrowhead framework, namely Eclipse Hono and Eclipse Ditto.

3.3.1.3 Eclipse Vert.x

Eclipse Vert.x implements the Reactor pattern, which employs a so-called event loop to process asynchronous events. In a server application, a new client connection is handled quickly by a main thread and potentially blocking operations, such as database requests, get resolved asynchronously by registering a callback function reference. Thus, the main thread remains responsive to new connection requests. Once a blocking operation finishes, the event and attached results are inserted into the event loop, which then takes care of the response processing. However, the most important rule when using Vert.x, is that there may be no blocking instruction executed by the event loop. Otherwise, new client connections could not be handled until the thread finished the blocking operation.
In contrast to similar reactive frameworks, such as Node.js which also features an event queue approach, but is limited to a single thread, Vert.x harnesses the full potential of multi core processors by applying a thread pool for event loops. This effective usage of resources is also reflected in the latest TechEmpower benchmarks [31].
In addition to the basic concurrency model, Vert.x provides further tools and patterns to structure reactive systems and organize their communication. The basic unit in which developers divide their code are so-called verticles. Verticles encapsulate application functionality, and are always executed on the same event loop, as depicted in Figure 4. Hence, the developer does not need to think about concurrent access to certain resources, as each event

loop is executed by its dedicated thread. In the default case, Vert.x starts two event loops per existing CPU, but this number is configurable. As reactive systems use message driven communication between their internal components, Vert.x provides the so-called event bus, which can be used to transmit data between the different verticles of an application.

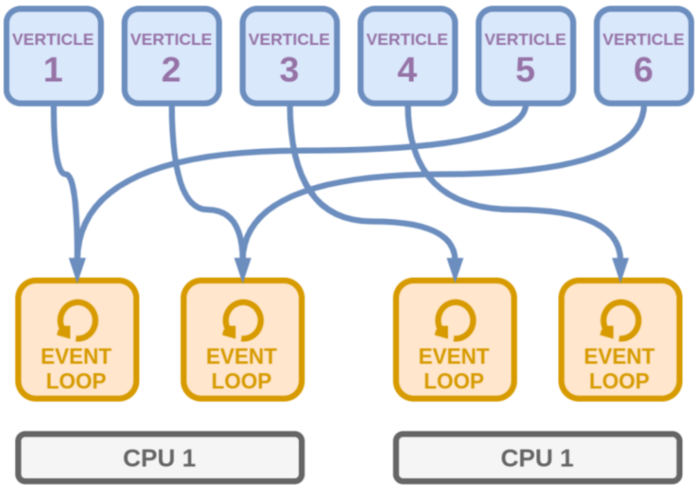


Figure 3.3 Verticle architecture in Eclipse Vert.x and reactive execution model on a dual core CPU with six deployed verticles.

Moreover, Vert.x offers a variety of native, non-blocking APIs to integrate with external systems, such as databases, message brokers, or web services. However, in case developers, in the absence of any alternatives, have to utilize a blocking library, Vert.x allows employing background threads to handle these operations.
One of the main benefits of using Vert.x, is that it is polyglot, which means that code can be written in Java, Kotlin, JavaScript, Groovy, Ruby and Scala.

3.3.1.4 Akka

In contrast to Vert.x, Akka implements the so-called Actor model [32], which was introduced by Hewitt et al. in 1973. An actor is the smallest building block in Akka, comparable to Vert.x verticles and encapsulates functionality of an application component and organizes its own state, which is not accessible from other actors. Each actor can

1. send messages to other actors,
2. receive messages by other actors,
3. and define mappings how to react to what kind of received message.

Upon message reception, actors pick an available thread from the thread pool. Also, they are guaranteed to run single-threaded, which is why they do not require parallelism measures, such as synchronization or locks. The messages they receive are processed in sequential order and their handling may result in passing messages to other actors. This implies again, that no blocking code may be executed in a message handler function, as otherwise the entire actor would be unable to process further messages. Similar to Vert.x, blocking APIs generally should be avoided, but can be invoked by passing them to dedicated worker threads.

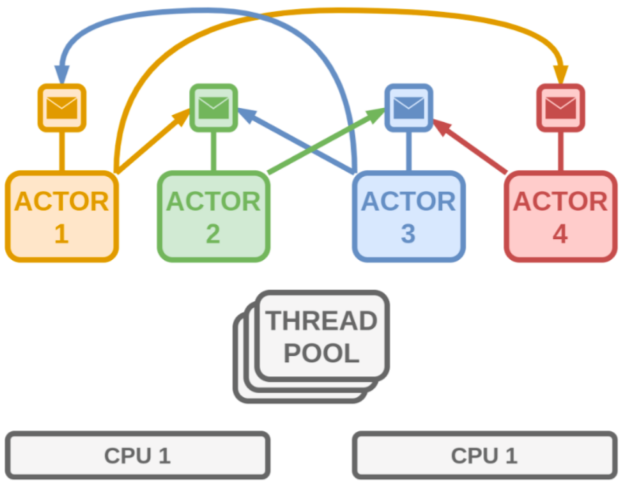


Figure 3.4 Akka Actor model architecture and communication flow.

In contrast to Vert.x, actors can spawn child actors which enables hierarchical system composition. Moreover, Akka also allows request-response based inter-actor communication patterns. However, these techniques come with loss of performance, as they involve Futures and the spawning of transient, internal actors.
Akka currently supports the programming languages Scala and Java and is used in the implementation of Eclipse Ditto. The underlying component, namely Eclipse Ditto, is built upon Akka. Hence, to seamlessly fit the Ditto microservices, the integration component is built of Akka actors.

3.3.1.5 IoT protocols

The following sections involve different application protocols, primarily used in IoT use cases, namely MQTT, AMQP, CoAP, and HTTP. This section provides a short explanation about these standards, including their characteristics and differences. More details can be found in [33], which this short comparison is based on.
MQTT is a many-to-many protocol, where each client connects via TCP with a broker, which then routes the incoming packages to the correct receiver. The sender directs its messages to a topic, which the recipients have to subscribe to. MQTT thus features a publish/subscribe based interaction pattern. It is designed to be implemented by constrained devices and also allows the usage of UDP on the underlying transport layer if necessary (MQTT-SN). Moreover, MQTT features TLS based encryption.
AMQP also involves a broker based infrastructure, but primarily focuses on reliability and security. In contrast to MQTT it allows publish/subscribe and request/response based interaction, and is less focused on resource limited devices, but more on high throughput. It uses TCP for transportation, TLS for security, and SASL [34] for authentication. Currently, there exist different versions of AMQP, which are not backwards compatible. RabbitMQ is a popular AMQP 0.9.1 message broker, whereas Apache Qpid implements AMQP in version 1.0.
CoAP is standardized by the IETF and the youngest of the presented protocol standards. It provides a REST like mechanism involving URIs, such as HTTP, but requires less resources and is thus an option for constrained devices to be implemented. In addition to request/response

based interactions, it allows a special form of the publish/subscribe pattern [35]. CoAP uses UDP for transportation and DTLS or IPsec for encryption.
HTTP actually is a web protocol and not primarily used in IoT contexts. However, due to its wide distribution it also finds application there. It features a request/response interaction pattern, uses TCP for transportation, and TLS for encryption. It is often used to realize RESTful services.

3.3.2 Targeted Framework integrations

The following section describes what integrations are planned and how they are approached.

3.3.2.1 Eclipse Hono as a Translation System

As the Arrowhead Framework comprises an extensible number of supporting core systems, this section proposes a new module for this collection, the Device Hub. This System provides a central messaging infrastructure that bundles and organizes all network traffic between interacting systems and harmonizes the heterogeneous protocol landscape. Hence, it solves a common challenge in an IoT infrastructure, which is incompatibility between different devices. The existing Arrowhead approach to this, is to deploy a translation system, which mediates these differences when necessary. The first use of Eclipse Hono in the Arrowhead context is therefore as a sort of translation system that allows translation between different protocols that are used within an Arrowhead cloud.

3.3.2.2 Eclipse Hono as an Arrowhead Device Hub

Hono's nature as a Device Hub in huge IoT deployments may also benefit a local cloud, as it provides a quality of service level in system interaction, which especially constrained devices cannot deliver. Also, the broker infrastructure benefits traffic efficiency, as it avoids polling and therefore reduces traffic load and frees up processing power on the device side.

3.3.2.3 Eclipse hawkBit as Arrowhead Configuration System

The Arrowhead Framework description proposes a so-called configuration system, to monitor the current device configuration and distribute changes [7]. These changes comprise new software rollouts of the actual application running on the device or the underlying operating system, and other configuration files.
As future Industrial IoT scenarios potentially involve hundreds of thousands of devices, each of which requires updates over time, a reliable and scalable solution is indispensable. Moreover, software or configuration updates might need to be distributed quickly, as they are intended to fix security breaches or bugs in highly critical components. If not eliminated in a short amount of time, these security risks potentially result in high costs. Hence, a configuration system solution would also need to force updates on certain devices as fast as possible.
Eclipse hawkBit provides a comprehensive service to control and monitor software rollouts and device configuration. For instance, that includes cascading software distribution, where the user organizes updates in groups of devices. Each group update gets triggered successively, depending on the successful deployment of the previous group. HawkBit also allows the integration of Content Delivery Networks (CDNs) to store high volumes of large files separately. This makes it a straightforward fit to handle the named requirements in the beginning.

3.3.2.4 A digital twin system based on Eclipse Ditto

In a manufacturing plant with a large number of devices on the shop floor, it might be beneficial to employ a so-called digital twin system. A digital twin represents all relevant characteristics of a physical instance in the real world. For example, a hypothetical weather station would encapsulate attributes, such as temperature, humidity, or wind speed. A potentially participating system, interested in those data, could then request the specified information and receive the current state. However, the digital twin system does not provide historic data, as this is not part of its functional scope.
Eclipse Ditto provides digital twin management capabilities as described above. It relies on the Akka framework and is structured in several microservices, interacting with each other and renders itself to a perfect fit as a digital twin system within the Arrowhead framework.

3.4 Status of Work

At the time of this report Bosch.IO GmbH has performed the initial analysis for all four targeted framework integrations and realized the implementations of Eclipse hawkBit as Configuration System. Additionally, also an initial feasibility study of Eclipse Hono as translation system was performed. Both results are discussed in what follows.

3.4.1 Eclipse hawkBit as Configuration System

As described previously, Eclipse hawkBit provides a comprehensive service to control and monitor software rollouts and device configuration.
The internal architecture of hawkBit consists of the following components.
Artifacts and metadata repository: Here, hawkBit stores the software artifacts, such as binaries or configuration files.
Direct Device Integration (DDI) API: The DDI is hawkBit's plain REST API, which enables devices to check for new updates. As it only allows for a polling based re- trieval of new artifacts, the user can adjust the frequency the devices should ask for updates. However, as polling, compared to pushing new updates, is highly inefficient, hawkBit provides the DMF API.
Device Management Federation (DMF) API: The DMF API is based on a RabbitMQ message broker instance. This API does not allow devices to connect directly, but enables developers to implement custom solutions, to integrate more protocols, e.g. LWM2M, which then provide device endpoints. Since RabbitMQ consists of an AMQP 0.9.1 messaging network, the DMF provides publish/subscribe capabilities, and hence allows efficient update procedures. However, the DMF not only supports software and configuration rollouts, but also some device management options, such as device creation and deletion.
Management API: The management REST API allows full control over hawkBit's con- figuration parameters. This includes the creation, deletion and modification of devices, software artifacts, rollout configurations and more.
Management UI: The management UI provides graphical elements for a user to interact with the management API. However, it could easily be replaced by a new custom UI, allowing the creation of an Arrowhead wide management UI.
One has the option to modify hawkBit's code base or expand the system as a whole by a wrapper. There remains a high risk that internal changes in the source code might be destroyed by or interfere with future releases. Also, an integration of the necessary changes into hawkBit's code in this case is not sensible, as it would narrow the applicable use cases and therefore lose its ability to be used outside an Arrowhead local cloud. Moreover, as hawkBit explicitly

provides an interface for third party implementations to integrate new protocols and mechanisms — the DMF API — the chosen solution features a wrapper-based approach.
To integrate hawkBit into the Arrowhead ecosystem, it has to be discoverable for all devices. Hence, a system administrator has to create entries in the service registry and the orchestration system. However, the wrapper first has to register at the service registry.

3.4.1.1 Wrapper architecture

Internally, the wrapper implementation is divided in three different Vert.x verticles, as depicted in Figure 3.5. The first one interacts with the RabbitMQ endpoint, which involves receiving and sending messages that either contain software update details sent to the devices or confirmation messages sent to hawkBit after an update went successful. Also, new devices, connecting for the first time with the wrapper, are registered in hawkBit, a feature that is provided by the DMF but not the DDI. The second verticle manages the device connection, currently via Websockets. However, this part of the application could easily be exchanged with any other protocol implementation, such as MQTT. In this scenario Websockets are chosen due to the fact that no broker infrastructure is needed and because the current Arrowhead implementation uses HTTP web requests in its services. The third verticle is responsible for all necessary interaction with the Arrowhead core systems, namely the authorization system and the service registry. This is necessary to register the wrapper as a service, as it would not be discoverable otherwise. Additionally, the wrapper periodically fetches the public authentication key from the authorization system, which is needed to verify the JSON Web Token, issued to the device during orchestration. The authentication flow is described in more detail below.
The verticles internally communicate using the Vert.x EventBus, which guarantees a high degree of decoupling, which is favorable when it comes to the integration of other protocol implementations for the second verticle, as mentioned earlier. Moreover, it separates all connector implementations from each other, making them all interchangeable, for instance in the case of an API change, either of the mandatory core systems or the AMQP messaging network. As RabbitMQ currently relies on AMQP 0.9.1, it is likely to upgrade to AMQP 1.0 in the future, which distinguishes itself drastically from its predecessor. In addition to the EventBus, the verticles very rarely use shared resources, except for the keystore which holds public and private key of the configuration system and is needed to decrypt the JWT payload. However, as different verticles run on different event queue threads, these shared resources need to be protected by locks.

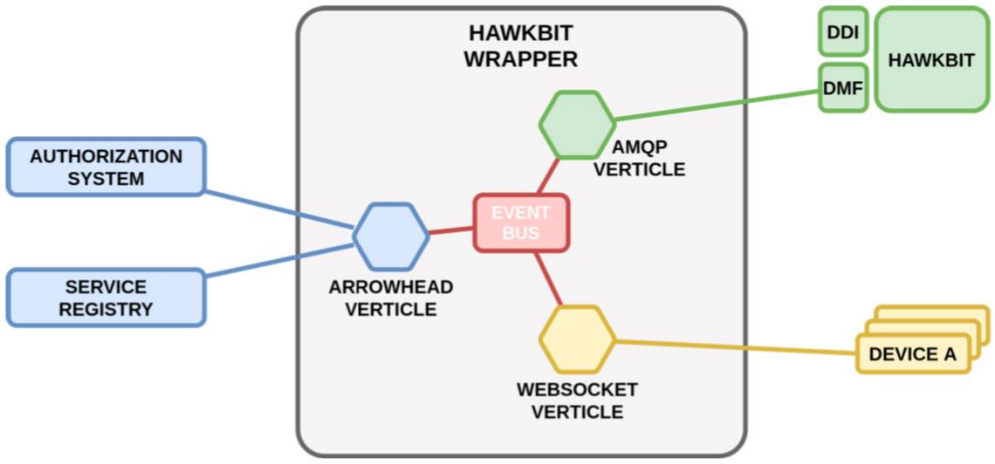


Figure 3.5 Architecture of the proposed configuration system. Three Vert.x verticles of a wrapper module communicate with the DMF API of Eclipse hawkBit (1), the Arrowhead mandatory core systems (2), and the respective devices (3).

3.4.1.2 Operating principle

Figure 3.6 depicts the interaction between all participants being a new device, the Arrowhead mandatory core services, and the configuration system wrapper together with a hawkbit instance. In the first step, the configuration system wrapper, as mentioned above, registers at the service registry. For this to be possible, the wrapper needs its own client certificate, just as any other system interacting with the core services. The client certificate is signed by the private key of the local cloud root certificate. Also, to verify the correctness of the server certificate the service registry requires a copy of the root certificate in a truststore. These steps need to be performed upfront by the local cloud administrator. However, as there might be hundreds of thousands of devices involved, an automated process, which for example requires local presence in a production facility, is advisable.

In addition to the setup on the Arrowhead side, the AMQP 0.9.1 connection module initializes a connection to the RabitMQ broker and registers a queue and an exchange. In AMQP 0.9.1, exchanges are used to publish message, whereas queues allow subscribing to incoming messages. Internally, queues are then linked to exchanges, which allows routing of messages to different queues. In this scenario however, one exchange is bound to one queue.

After the bootstrapping process completed, the system administrator has to define orchestration and authorization rules, mapping devices to the configuration system and giving them permission to request configuration updates. Subsequently, devices can request configuration orchestration at the orchestration system, which will provide them with the correct protocol, system address, port, service URI and a JSON Web Token (JWT) [22]. This nested JWT, allows the service consumer to authorize at a service producer.

Subsequently to the orchestration, the consumer system attempts to open a Websocket connection to the configuration wrapper. During the Websocket handshake, the config- uration system then checks the JWT using its own private key for decryption and the authorization system public key for validation. The exact validation, as performed by the configuration system is depicted in Listing 3.1. The transmitted token is validated, and the included claims are extracted. It is also important to note that the validation operation has to be performed by a separate Vert.x background thread, as it otherwise might block the event loop. Therefore, the Vert.x executeBlocking API is called.

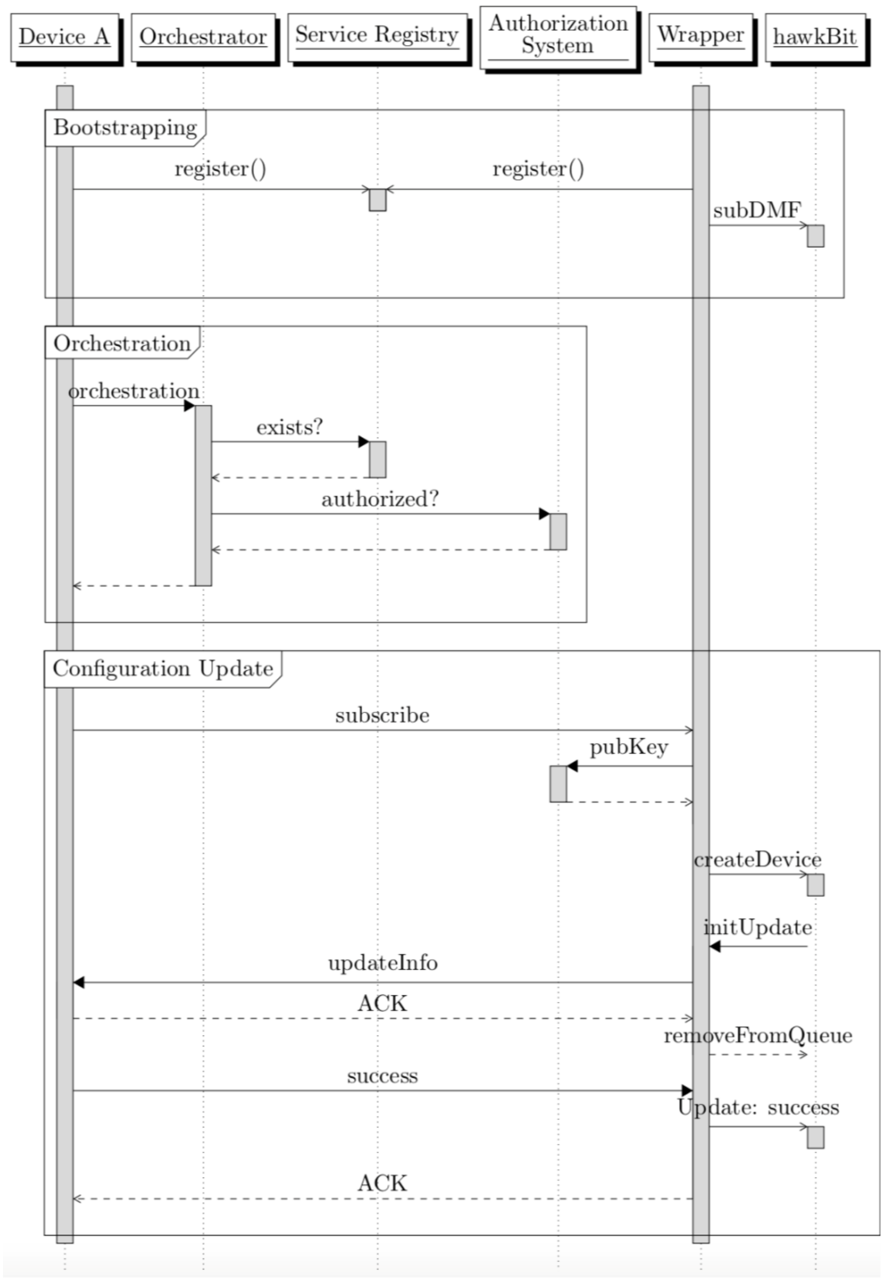


Figure 3.6 Interaction between a potential device requesting configuration updates, the Arrowhead core systems, and a configuration system based on hawkbit.

```
public Future<JwtClaims> validateTokenAndRetrieveClaims(String token,
    PublicKey authorizationPubKey, PrivateKey providerPrivKey) {
    Promise<JwtClaims> promise = Promise.promise();

    vertx.executeBlocking(innerPromise -> {
        try {
            final JwtConsumer jwtConsumer = new JwtConsumerBuilder().
    setRequireJwtId()
                    .setRequireNotBefore()
                    .setEnableRequireEncryption()
                    .setEnableRequireIntegrity()
                    .setExpectedIssuer("Authorization")
                    .setDecryptionKey(providerPrivKey)
                    .setVerificationKey(authorizationPubKey)
                    .setJwsAlgorithmConstraints(new AlgorithmConstraints(
    ConstraintType.WHITELIST, AlgorithmIdentifiers.RSA_USING_SHA512))
                    .setJweAlgorithmConstraints(new AlgorithmConstraints(
    ConstraintType.WHITELIST, KeyManagementAlgorithmIdentifiers.
    RSA_OAEP_256))
                    .setJweContentEncryptionAlgorithmConstraints(new
    AlgorithmConstraints(ConstraintType.WHITELIST,
    ContentEncryptionAlgorithmIdentifiers.AES_256_CBC_HMAC_SHA_512))
                    .build();

            JwtClaims claims = jwtConsumer.processToClaims(token);

            innerPromise.complete(claims);
        } catch (Exception e) {
            innerPromise.fail(e);
        }
    }, (AsyncResult<JwtClaims> res) -> {
        if(res.succeeded()) {
            promise.complete(res.result());
        } else {
            promise.fail(res.cause());
        }
    });

    return promise.future();
}
```

Listing 3.1 Arrowhead token validation implementation.

Afterwards, the connection gets stored in a hash map, using the consumer system ID as key. The system ID is encapsulated in the JWT payload claims. In case the consumer connects for the first time, the wrapper creates a new device in hawkBit. This is legitimate, as after successful orchestration and authentication, the configuration system rightly assumes that the consumer is authorized to receive configuration updates. Now, the consumer device permanently is connected with the configuration wrapper, ready to receive software updates. This illustrates the necessity of a reactive programming approach, as it would be unmaintainable to keep a thread for each connection, with a large number of devices connected permanently.
After the local cloud administrator initiates a software update for a system or a group of systems, hawkBit publishes messages into the RabbitMQ messaging network. The configuration wrapper, in particular the AMQP connector verticle, which is subscribed to these notifications, then opens the messages and categorizes them, depending on their topic header.

Afterwards, the payload is forwarded to the WebSocket verticle using the event bus. However, it is important that the message reception is not immediately acknowledged to the RabbitMQ broker, but only after the device successfully received it, which is why the AMQP verticle waits for a response on the event bus. Otherwise, the wrapper implementation could not guarantee the successful delivery, as the message would be lost, and the update could not be triggered.
Upon successful reception of the update request message, the WebSocket verticle, which is listening on the event bus for incoming messages from the AMQP verticle, extracts the recipient system ID from the message body. In case the ID does not exist in the hash map of held WebSocket connections or the transmission fails, the verticle reports back the error through the event bus. Otherwise the device gets notified about the new notification. However, hawkBit does not just deliver update notifications and provides software artifacts, but also tracks the rollout state. This implicates that the device needs to provide feedback to hawkBit. Hence, the wrapper implementation also comprises a back channel for the systems that notify hawkBit upon successful or failed software update. This information is crucial for the system administrator and also features hawkBit's cascading rollouts mechanism, which relies on the information of active rollout processes to potentially stop further updates and request manual intervention.
Table 2.2 lists all available hawkBit services usable via the DMF API. The upper part of the table presents all requests directed at hawkBit, the lower part views all commands sent by hawkBit to a certain device. The wrapper supports a subset of these interactions, which are highlighted in bold letters. For a fully functional and production-ready implementation, eventually all services would need to be connected. However, for a first proof of concept, the implemented subset is sufficient, as it covers the basic functionality necessary to integrate new devices, guarantee they are authorized, and perform a complete software rollout, including the processing of status updates coming from the devices. It is planned to add the other commands later.
Using the UPDATE ACTION STATUS message, the device continually informs hawkBit about the update process. Status changes occur, for instance, once the download of a software artifact started, finished, or after the installation completed successfully. The device autonomously decides, which update information to provide, as not all intermediate steps might deem necessary to transmit. However, the more fine-grained the reporting and error tracing are, the easier becomes the troubleshooting.
To guarantee that the list of held WebSocket connections within the hash map always matches the set of real connections, a registered callback function block is responsible to remove inactive systems from the data structure. This piece of code is invoked by the underlying WebSocket library after the keepalive timeout was exceeded. This mechanism is not to be confused with the optional TCP keepalive feature. WebSocket keepalive messages guarantee that both sides — client and server — are still available by sending a ping message that has to be acknowledged by the other side with a pong message in a certain amount of time. Also, a system might attempt to connect to the configuration wrapper more than once, which is technically possible, but would be absolutely inconsistent in the Arrowhead Framework, as one system can request exactly one configuration. In this case, the controller rejects the WebSocket handshake, as there is no reason to connect for a second time.

3.4.1.3 Evaluation

The described implementation basically fulfills the entire spectrum of functionality of an Arrowhead configuration system. It allows creating automated software and configuration

rollouts and monitoring their success. Thanks to the supplementary wrapper implementation, it also seamlessly integrates into Arrowhead's orchestration and authorization mechanism. However, the hawkBit API provides more endpoints, which have to be implemented too, if usage scenarios require those.
Due to its modular architecture, the wrapper implementation allows for the replacement of each of the three microservices. The endpoint component, handling all interaction with the devices, currently supports WebSocket connections, as there is no intermediary broker infrastructure necessary, and WebSockets allow bidirectional communication. Also, Vert.x provides a native and non-blocking WebSocket API. A future application might potentially support a different protocol, such as LWM2M, which is based on CoAP and specifically designed for device management and configuration purposes.

Name	Description
THING_CREATED	Creates a new device. All incoming messages will be sent to the AMQP exchange in the reply_to header.
THING_REMOVED	Removes an existing device in hawkBit.
UPDATE_ATTRIBUTES	Allows creating/updating user-defined attributes for a device
UPDATE_ACTION_STATUS	Indicates that a device began a new step in the update process or finished it (either succesful or unsuccessful). Valid values are DOWNLOAD, DOWNLOADED, RETRIEVED, RUNNING, FINISHED, ERROR, WARNING, CANCELED, or CANCEL_REJECTED.
PING	Allows a device to check whether the DMF API is available
CANCEL_DOWNLOAD	Notifies a device that the current update process should be canceled, if possible.
DOWNLOAD_AND_INSTALL DOWNLOAD	Initiates a new update process at the respective device, providing all necessary information, such as the download address of the configuration or software artifact.
MULTI_ACTION	Alternative endpoint to DOWNLOAD_AND_INSTALL, DOWNLOAD, and CANCEL_DOWNLOAD
THING_DELETED	Informs about a manual deletion of a specific hawkBit device, possibly by a system administrator.
REQUEST_ATTRIBUTES_UPDATE	HawkBit requests re-transmission of device attributes
PING_RESPONSE	HawkBit response to a PING message

Table 3.2 Provided hawkBit services via the DMF API. The wrapper implementation supports a subset which is highlighted in bold letters.

3.4.2 Eclipse Hono as Translation System

Due to Hono's diverse protocol adapters, it might also fit as foundation for an Arrowhead translation system. The translation system offers translation capabilities to connect systems, which otherwise could not communicate with each other, due to differing supported application layer protocols. In the best case the translation happens transparent for the involved services.

The applicability of Hono as Translation system was evaluated with Proof of Concept implementations and analyzed with respect to its overall fit.

3.4.2.1 Functionality of a Hono-based translation system

The Arrowhead translation system functions as an intermediary and transient layer between two communication partners, the producer and the consumer. To let Hono perform this role, a mirroring service could connect to the northbound API to which normally business applications use to interact with the devices through Hono. This architecture with the mirroring service (translation engine) is depicted in Figure 3.7. Every incoming message would then need to be unwrapped and transmitted to the correct receiver by this service. The producer and consumer could easily connect to the protocol adapters and the mirroring wrapper module would forward every message to the correct receiver.

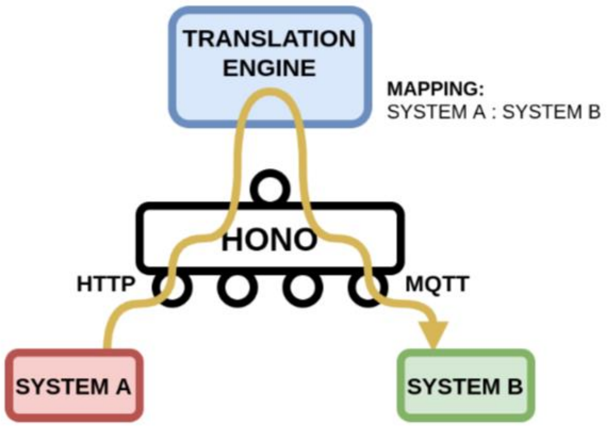


Figure 3.7 Architecture concept of a Hono based translation system.

Figure 3.8 depicts the entire communication process. The consuming service (App System A) requests orchestration at the orchestration system. As the service registry stores metadata, such as which protocol a service uses, the orchestrator then determines whether a translation is necessary. After checking with the service registry and the authorization system, to ensure the consumer exists and is authorized to consume the producing service (App System B), the orchestrator requests a translation process at the translator system. Internally, Hono registers two new devices and creates credentials for them, which it hands over to the orchestrator. In order to enable the connection between the orchestrator and the translation system, the existing orchestrator reference implementation would need to be modified accordingly, as it is not aware about translation operations yet. In the initial Arrowhead definition [7], the translation system provides an API, which allows to instantiate a new transient translation unit. This translation unit then temporarily organizes a particular translation procedure between two systems and vanishes afterwards. As Hono does not fit the requirement of being deployed for each translation session, the wrapper rather provides the possibility to configure a translation flow between two participants instead. This configuration would involve the creation of two devices and their credentials. Here, the device abstractions in the Hono context are repurposed to allow Arrowhead systems to connect.

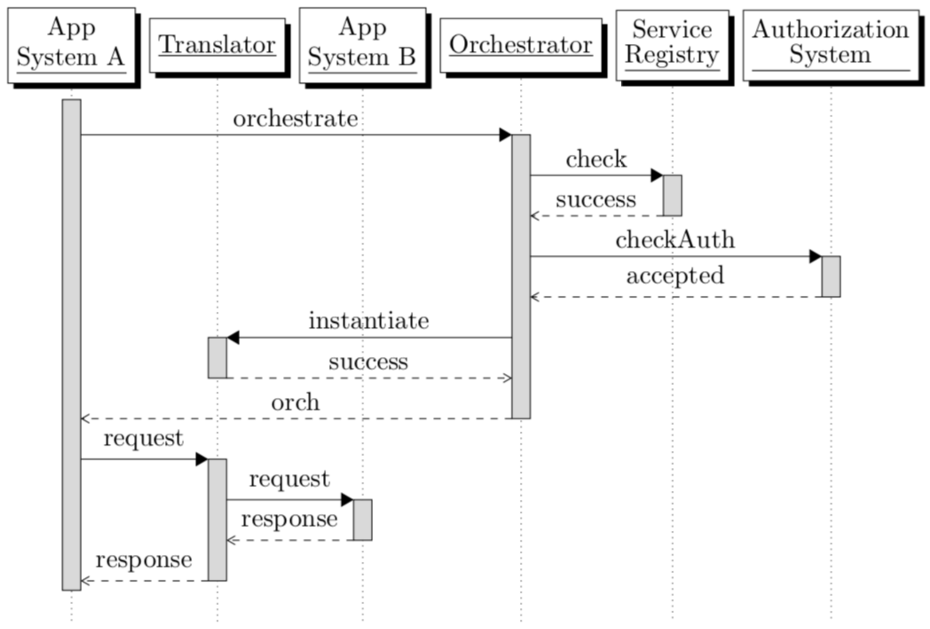


Figure 3.8 Translation system communication pattern

After the setup completed, the orchestrator returns the orchestration result to the consumer system, including the related credentials. The consuming service then connects to the correct protocol adapter, using its Hono device credentials. The translator, acting as a proxy, consumes the producing service and forwards the results to the actual consumer.

3.4.2.2 Evaluation

Looking at the described interaction flows, it becomes clear that Hono does not fit as an ideal translation system foundation, because it is not capable of transparently translating between publish/subscribe and request/response based protocols. Figure 3.9 shows the four different use cases concerning the translation between HTTP (request/reponse) and MQTT (publish/subscribe), where red paths are indicating systemic shortcomings.

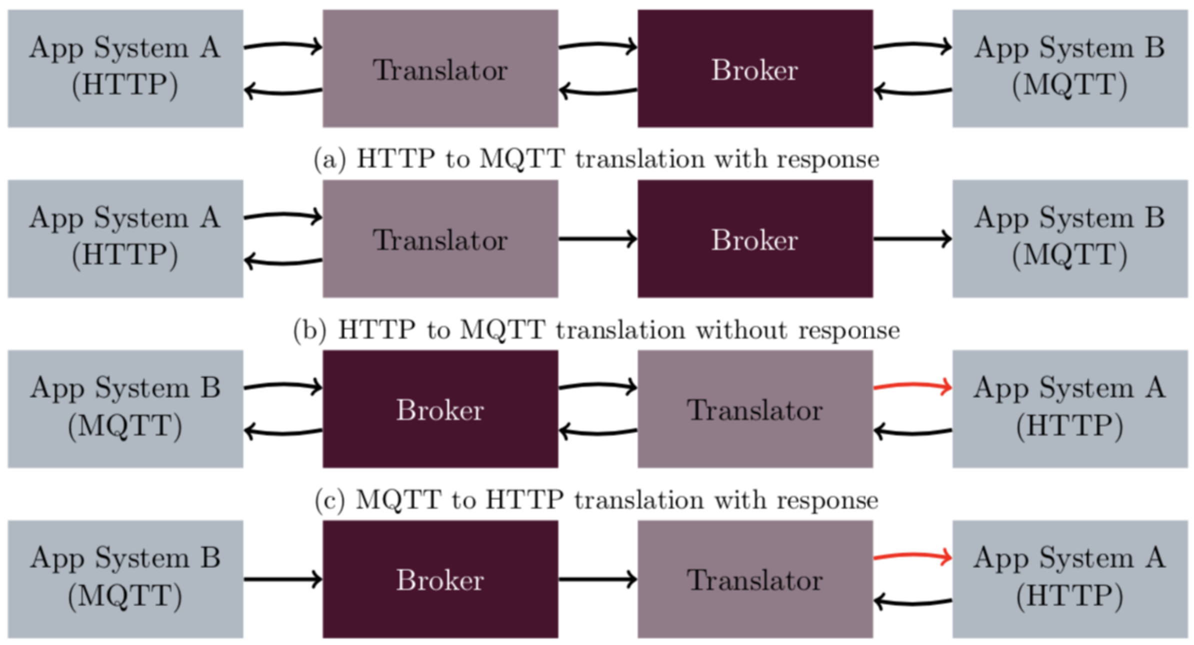


Figure 3.9 Hono's translation capabilities come with systemic flaws in regard to protocol translation between publish-subscribe and request-reponse based protocols.

Figure 3.9a depicts a HTTP system consuming a MQTT service, expecting a synchronous response. In Figure 3.9a, in contrast, App System A does not expect a response from the producer. Both use cases generally could be performed by a Hono based translator. However, App System B would need to connect directly to Hono's internal MQTT API, which might not be desirable for use cases without any translation system. This is due to the fact that Hono does not provide the possibility to use an external MQTT broker, subscribe to a specific topic, and forward the incoming messages.
Figure 3.9c and Figure 3.9d present the reverse use case. In this scenario, the MQTT consumer connects to a broker, either just by subscribing to a topic or additionally publishing a message to another topic. The mirroring service then retransmits the message to the producer and waits for the synchronous response, which it publishes using the broker. The MQTT consumer then receives the response. However, Hono is not capable of sending HTTP requests over its protocol adapters as it lacks an HTTP client implementation. Hence, the producer would need to send a request, indicating over a Hono-specific TTL- header that it wants to wait for a response. However, this fundamentally ignores the consumer/producer concept where the consumer initiates the transmission.
Both presented cases — MQTT to HTTP and vice versa — reveal profound structural drawbacks in a translator implementation. The absence of client implementations only allows for very specific use cases, where no separate MQTT broker infrastructure is involved and an HTTP producer system connects to the translator in advance, waiting for potential responses. Moreover, Hono's specific terminology restricts the potential applications or at least complicates usability. For instance, if a system sends data via MQTT it has to comply with Hono's definite topic structure, starting with telemetry/.... Even though this is primarily a convenience argument, it also complicates the replacement of a different translator system, as application systems have to adapt to these specifics and therefore have be aware of the ongoing

translation which should be transparent to the involved endpoints to create a higher degree of interoperability.
Moreover, a Hono based translation system only fulfills the Arrowhead requirements in regard to authentication and authorization, if one chooses the client certificate-based authentication in combination with the auto provisioning feature of Hono. The credentials-based authentication also supported in Hono would require an intermediate step to register each participant and distribute the credentials.
Additionally, as described above, the translation system needs to maintain a participant mapping for as long as the translation procedure continues. As there might arise situations with one participant being involved in more than one translation flow, the translation system also needs to be capable of distinguishing between these flows. This could either be done by linking each flow with a unique procedure number, or by generating two new devices for each translation. The latter option does not require the communicating systems to be aware of the translation process, but rather provides a transparent API, where the consuming system cannot distinguish between a translated and an untranslated consumption, as it does not need to include the procedure number. As this simplifies the entire process, this would be the preferred approach.
Considering the above-mentioned points, Hono does offer translation capabilities, and certainly helps to connect devices in a heterogeneous protocol landscape but is not compliant to the Arrowhead translation system definition. So this work will not further enlarged in favor of other, more suitable approaches (e.g. the Translator system presented in Chapter 15).

4 Eclipse Kura and Kapua

This chapter illustrates the integration of Eclipse Kura and Kapua with the Arrowhead Framework. The two projects have been conceived, developed and promoted respectively by Eurotech and Eurotech/Red Hat in the context of the Eclipse Foundation IoT Initiative, where Eurotech is working with Cloudera, Red Hat, and others to develop key IoT runtimes and other enabling technologies that will allow the creation of end-to-end integrated and open IoT architectures.
Eclipse Kura™ is an extensible open source IoT Edge Framework based on Java/OSGi. Kura offers API access to the hardware interfaces of IoT Gateways (serial ports, GPS, watchdog, GPIOs, I2C, etc.). It features ready-to-use field protocols (including Modbus, OPC-UA, S7), an application container, and a web-based visual data flow programming to acquire data from the field, process it at the edge, and publish it to leading IoT Cloud Platforms through MQTT connectivity.
Eclipse Kapua™ is a modular IoT cloud platform to manage and integrate devices and their data. A solid integrated foundation of IoT services for any IoT application.
The projects represent the two main counter parts of the IoT integration solution that is adopted in use case 8.3 “SoS engineering of IoT edge devices (Condition Monitoring)”
and use case 8.4 “SoS engineering of IoT edge devices (Smart Home)”.

4.1 Rationale for Integration

The integration of Eclipse Kura and Kapua with the Arrowhead Framework will introduce fleet-level functionalities that will allow to monitor, from a centralized location, a set of AF Local Clouds remotely deployed and running on the field. Moreover, Kura will improve the capabilities of the AF Local Cloud in terms of data collection, edge computing and data delivery to existing cloud platforms.

4.2 Technical Details

The envisioned architecture has been organized in two different levels, according to the two main potential usage of the Arrowhead Framework. The first level refers to the edge, where a multiservice gateway or an industrial edge controller hosts both Eclipse Kura and the AF-based Local Cloud. While the second level refers to the AF-based Enterprise Level Cloud that runs on the enterprise side together with Eclipse Kapua.

The integration of Eclipse Kura with the Local Cloud will extend the Local Cloud capabilities in terms of data collection, data processing and connectivity. Indeed, Kura supports several field protocols, data collection mechanisms and the hardware abstraction layer simplifies the process of extending them, supporting new protocols, etc. Eclipse Kura is also a programming environment that wraps the complexity of low-level device management with high level constructs, providing application-oriented abstraction levels that translate in services intended to simplify and speed-up the software development.

Eclipse Kura represents the edge-side enabler for the fleet management and provides an efficient and easy way to collect data from the Local Cloud, process data in the Local Cloud and, only when it is required, to send raw or processed data to the cloud or a data center using a MQTT connection, natively optimized for telemetry and command/control.

Eclipse Kapua is autonomously capable to provide the complementary functionalities required to monitor the remote fleet of Local Clouds. An efficient and scalable MQTT broker collects information from the fleet, a rule engine process it and a non-relational database, optimized for time series, can efficiently store it, making it available for added-value services. Although Kapua provides this possibility autonomously, without the need of the Arrowhead Framework, it is extremely more valuable to integrate it with the framework and publish/consume AF services. Indeed, thanks to the integration, the interoperability level increases, the number of accessible services increases, it is possible to take advantage of the service-oriented architecture of the framework and simplify the design and development of domain-specific and, in particular, of cross-domain applications

A Fleet Monitoring Dashboard is an example of this kind of application: it extends Eclipse Kapua to display Local Cloud specific information that allows a single operator to monitor a fleet of Local Clouds deployed on the edge, independently from the geographical location, application context, specific business logics, etc. The Fleet Monitoring Dashboard adopts a Service Oriented architecture and can collect the information from the fleet of Local Clouds both from Eclipse Kapua or through the services published by Eclipse Kapua on the Enterprise Level AF Cloud. Following this approach, it will be also possible to develop separate applications for specialized operators (e.g. the installer or the maintenance operator) or for the end-user.

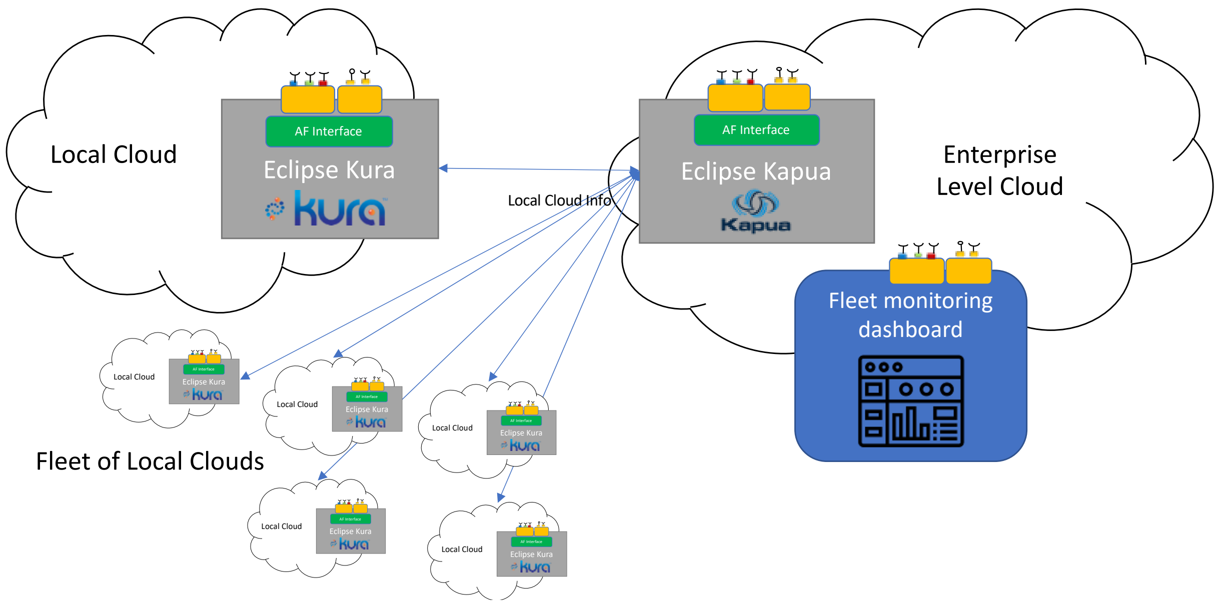


Figure 4.1 Integration of Eclipse Kura and Kapua.

4.3 Integration in Arrowhead Framework

The integration of Eclipse Kura will be based on the internal service-oriented abstraction layer available in Kura itself, a layer that allows to hide all the low-level technical details regarding hardware functionalities, data acquisition, data processing, connectivity, etc. Kura provides a Java/OSGi-based container for M2M applications running in service gateways and is specifically conceived for the edge computing and IoT domains. Thanks to a service-oriented architecture, Kura provides or aggregates open source implementations for the most common services needed by M2M applications, implemented as configurable OSGi Declarative Service exposing service API and raising events.

Kura includes the following set of services:

- I/O Services
- Data Services
- Cloud Services
- Configuration Service
- Remote Management
- Networking
- Watchdog Service
- Web administration interface

The integration will be based on the extension of the data publishing service, data acquisition service and control service, that will provide specific REST APIs and will be published in the AF as AF services. Data acquisition services are intended for data acquisition from the field. Data publishing service is responsible for the information stored at cloud level. This service receives data from the data acquisition services and publish them on a specific repository (e.g. a cloud platform, a remote database, etc.). The service abstracts from the specific communication protocol adopted to connect to the remote repository. Finally, the control

service is responsible for the actuation of commands on Kapua side and vice versa, from Kapua to Kura.
With a similar approach, the monitoring services will be provided in the form of a REST API and will be published in the AF as AF services.

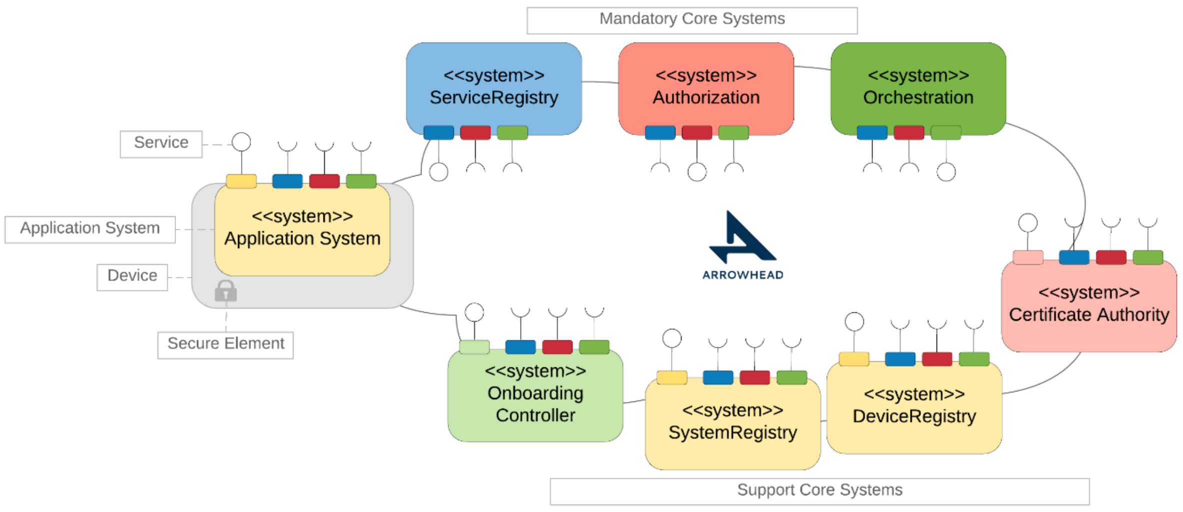
4.4 Status of Work

The definition of the overall architecture clarifying the role of the Arrowhead Framework, of Eclipse Kura and Kapua, their integration and their use in the use cases has been concluded.
The definition of the integration approach between the Arrowhead Framework and Eclipse Kura/Kapua, considering their different roles, functionalities and internal architecture has been concluded.
The design and development of the integration between the Arrowhead Framework, Eclipse Kura and Kapua have been just started and are ongoing at the time of publication of this deliverable.

5 Onboarding Procedure

5.1 Rationale for Integration

The onboarding procedure shown in Figure 5.1Figure 5.1 is needed when a new device produced by any vendor (e.g. Siemens, Infineon, Bosch, etc.) wants to interact with the Arrowhead local cloud. To assure that the local 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, containing a secure element (e.g. TPM), to its hosted application systems and their services. Thus, the onboarding procedure makes possible that devices, systems and services are authenticated and authorized to connect to the Arrowhead local cloud.



Mandatory Core Systems
<<system>> ServiceRegistry
<<system>> Authorization
<<system>> Orchestration
Service
Application System
<<system>> Application System
Device
Secure Element
ARROWHEAD
<<system>> Certificate Authority
<<system>> Onboarding Controller
<<system>> SystemRegistry
<<system>> DeviceRegistry
Support Core Systems

Figure 5.1: Onboarding Procedure

5.2 Technical Details

As shown in Figure 5.1, the onboarding procedure involves a number of Arrowhead core systems. In the following we provide an update of Arrowhead support core systems involved in the onboarding procedure, which are developed by FB.

5.2.1 Onboarding Controller System

The Onboarding Controller is a system at the edge of the Arrowhead local cloud, which is not part of the local cloud chain of trust. Thus, the Onboarding Controller system (i) is the first entry point to the local cloud, e.g. accepts all devices to connect via the Onboarding service, (ii) has a certificate for the https communication with the device, and (iii) (optionally) the certificate is provided by a public CA (e.g. Verisign). On success, the system provides: (i) an Arrowhead issued “onboarding" certificate, and (ii) the endpoints of the DeviceRegistry, SystemRegistry, ServiceRegistry and Orchestrator systems. As shown in Figure 5.2, the Onboarding Controller system consumes the ServiceDiscovery, Orchestration, AuthorizationControl and SignCertificate services and provides the Onboarding service.

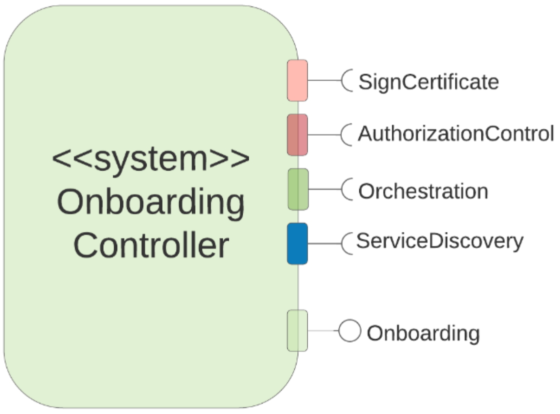


Figure 5.2: Onboarding Controller System

Figure 5.3 shows the use cases that represent the actors and their interaction with the Onboarding Controller system. The actors can be devices with different credentials: (i) device with a preloaded Arrowhead certificate, (ii) device with a manufacturer certificate, and (iii) device with a shared secret.

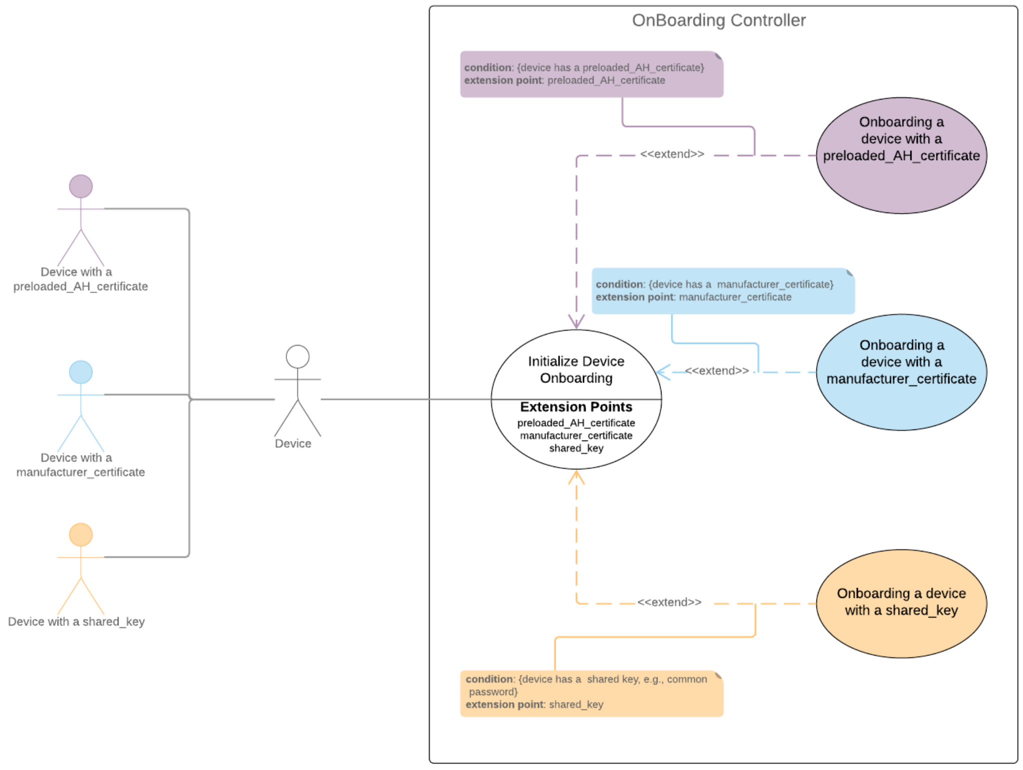

OnBoarding Controller
condition: {device has a preloaded_AH_certificate}
extension point: preloaded_AH_certificate
<<extend>>
Onboarding a device with a preloaded_AH_certificate
condition: {device has a manufacturer_certificate}
extension point: manufacturer_certificate
Initialize Device Onboarding
Extension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

Figure 5.3: Onboarding Controller Use Cases

5.2.2 DeviceRegistry System

The DeviceRegistry system is used to provide a local cloud storage holding the information on which devices are registered within a local cloud, meta-data of these registered devices, including a list of the systems that are deployed in each of them. The DeviceRegistry system holds for the Arrowhead local cloud unique device identities. The DeviceRegistry system shall be accessible using different SOA protocols (e.g. REST, CoAP, MQTT). As shown in Figure 5.4, the DeviceRegistry system consumes the three mandatory core services of Arrowhead, the SignCertificate service provided by CA and provides the DeviceDiscovery service.

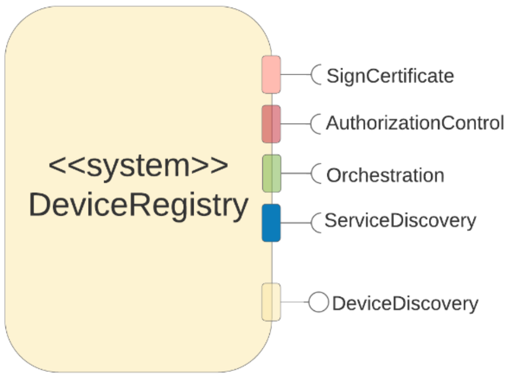


Figure 5.4: DeviceRegistry System

Figure 5.5 shows the use cases that represent the actors and their interaction with DeviceRegistry.

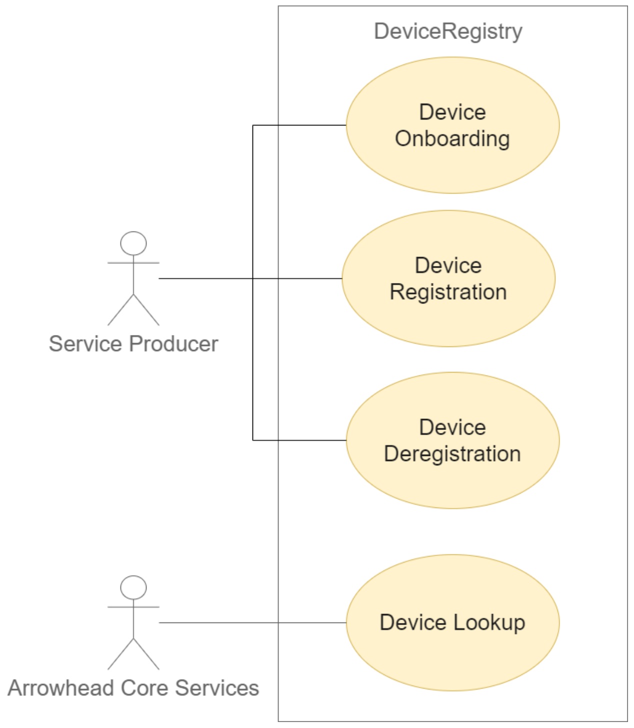


Figure 5.5 DeviceRegistry Use Cases

5.2.3 SystemRegistry System

The SystemRegistry system is used to provide a local cloud storage holding the information on which systems are registered within a local cloud, meta-data of these registered systems and the services these systems are designed to consume. The SystemRegistry holds for the Arrowhead local cloud unique system identities for systems deployed within it. As shown in Figure 5.6, the SystemRegistry system consumes the three mandatory core services of Arrowhead, the SignCertificate service produced by CA, and produces the SystemDiscovery service.

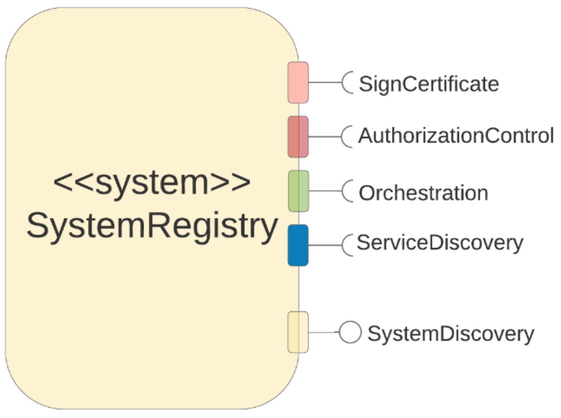


Figure 5.6: SystemRegistry System

Figure 5.7 shows the use cases that represent the actors and their interaction with SystemRegistry.

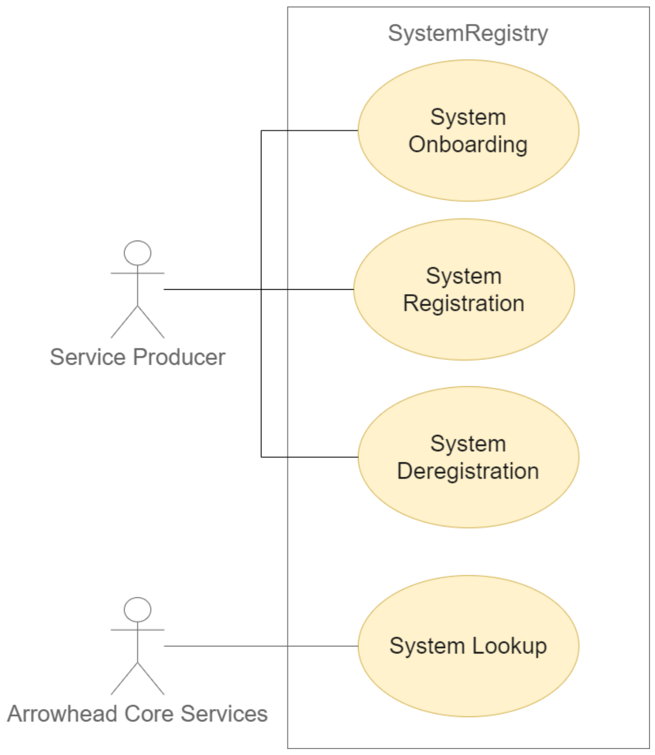


Figure 5.7: SystemRegistry Use Cases

The above mentioned registries, use a basic three-tier architecture: (i) The presentation tier named *Controller.java, which transforms the view into domain specific objects and vice versa. Each RegistryController contains three functions: *query*, which searches for an entity with e.g. a specific name (common name as shown in its certificate), *register*, which stores an entity in the database, and *unregister*, which removes an entity from the database. (ii) The application tier is named *Service.java, which is responsible for any business logic and extensive validation. (iii) The data tier is taken from the Arrowhead common project and is named *Repository.java. The Repository classes are generic interfaces using Spring Boot Data for implementation, They allow to deal with any entity in the Arrowhead code. Arrowhead uses OpenAPI (formerly known as swagger) to enrich the documentation of its REST methods.

5.3 Integration in Arrowhead Framework

The sequence diagram in Figure 5.8 shows the interaction of a new device with the Arrowhead local cloud during the onboarding procedure.

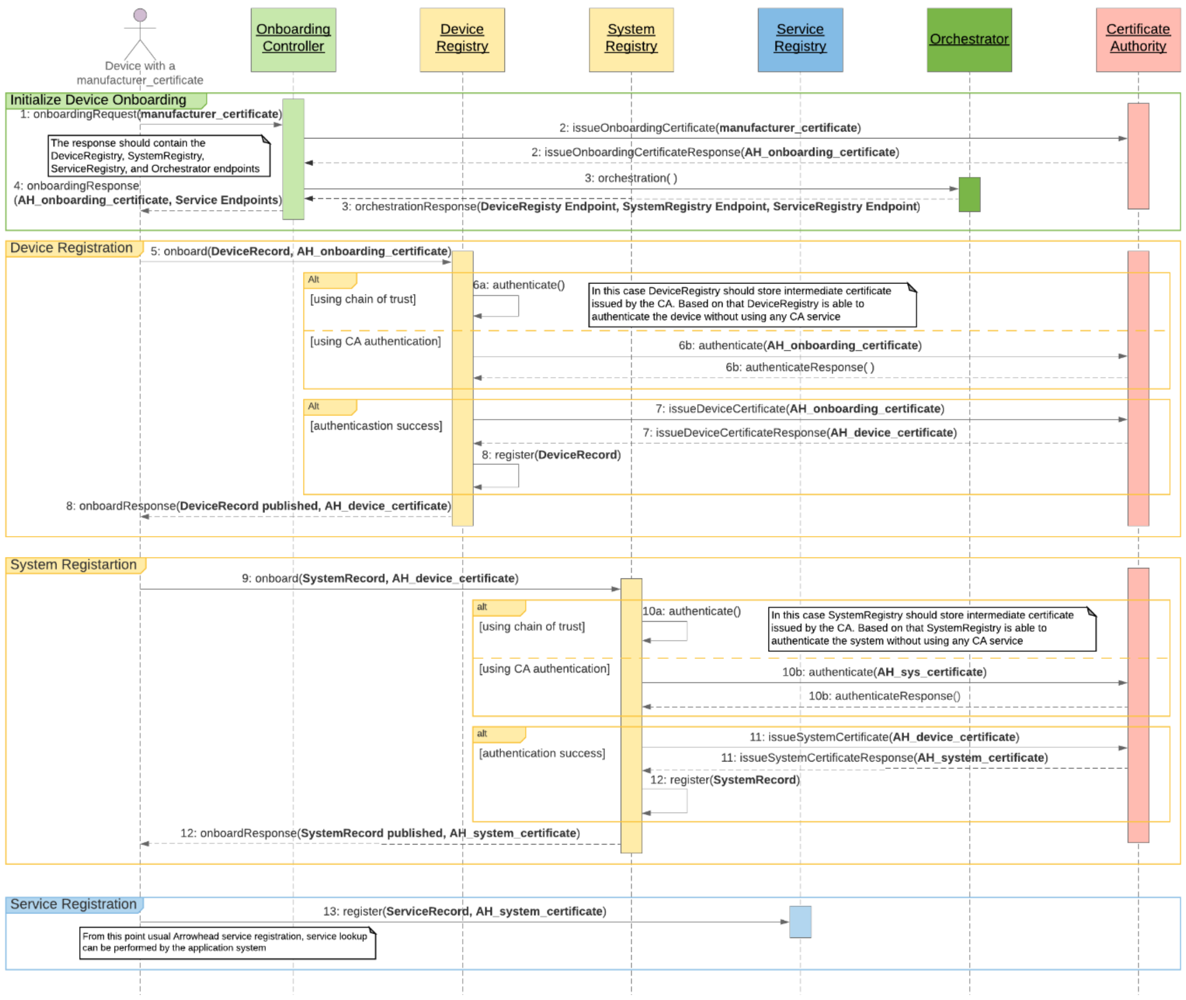

Device with a manufacturer_certificate
Onboarding Controller
Device Registry
System Registry
Service Registry
Orchestrator
Certificate Authority
Initialize Device Onboarding
1: onboardingRequest(manufacturer_certificate)
2: issueOnboardingCertificate(manufacturer_certificate)
The response should contain the DeviceRegistry, SystemRegistry, ServiceRegistry, and Orchestrator endpoints
2: issueOnboardingCertificateResponse(AH_onboarding_certificate)
3: orchestration( )
4: onboardingResponse (AH_onboarding_certificate, Service Endpoints)
3: orchestrationResponse(DeviceRegisty Endpoint, SystemRegistry Endpoint, ServiceRegistry Endpoint)
Device Registration
5: onboard(DeviceRecord, AH_onboarding_certificate)
Alt
[using chain of trust]
6a: authenticate()
In this case DeviceRegistry should store intermediate certificate issued by the CA. Based on that DeviceRegistry is able to authenticate the device without using any CA service
[using CA authentication]
6b: authenticate(AH_onboarding_certificate)
6b: authenticateResponse( )
Alt
[authenticastion success]
7: issueDeviceCertificate(AH_onboarding_certificate)
7: issueDeviceCertificateResponse(AH_device_certificate)
8: register(DeviceRecord)
8: onboardResponse(DeviceRecord published, AH_device_certificate)
System Registartion
9: onboard(SystemRecord, AH_device_certificate)
alt
[using chain of trust]
10a: authenticate()
In this case SystemRegistry should store intermediate certificate issued by the CA. Based on that SystemRegistry is able to authenticate the system without using any CA service
[using CA authentication]
10b: authenticate(AH_sys_certificate)
10b: authenticateResponse()
alt
[authentication success]
11: issueSystemCertificate(AH_device_certificate)
11: issueSystemCertificateResponse(AH_system_certificate)
12: register(SystemRecord)
12: onboardResponse(SystemRecord published, AH_system_certificate)
Service Registration
13: register(ServiceRecord, AH_system_certificate)
From this point usual Arrowhead service registration, service lookup can be performed by the application system

Figure 5.8: Onboarding Procedure Sequence Diagram

5.4 Status of Work

All systems involved in the Onboarding Procedure are released systems integrated in the Arrowhead GitHub master branch: https://github.com/arrowhead-f/core-java/tree/master

6 Monitoring and Standard Compliance Verification

6.1 Rationale for Integration

Monitoring and Standard Compliance Verification (MSCV) is used to monitor and verify if a new device, including its software systems and hosting services, that wants to interact with the Arrowhead local cloud fulfills the requirements of a specific standard.

6.2 Technical Details

The MSCV system shown in Figure 6.1 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 services in ServiceRegistry.

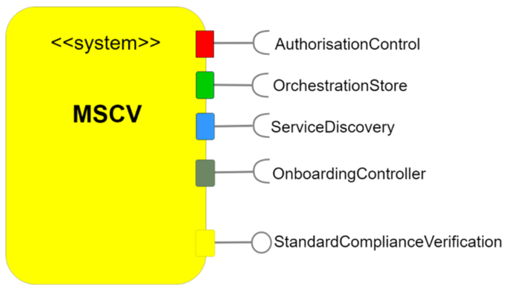


Figure 6.1 MSCV System

6.3 Integration in Arrowhead Framework

The MSCV system will be invoked during the Onboarding procedure as shown in Figure 6.2.

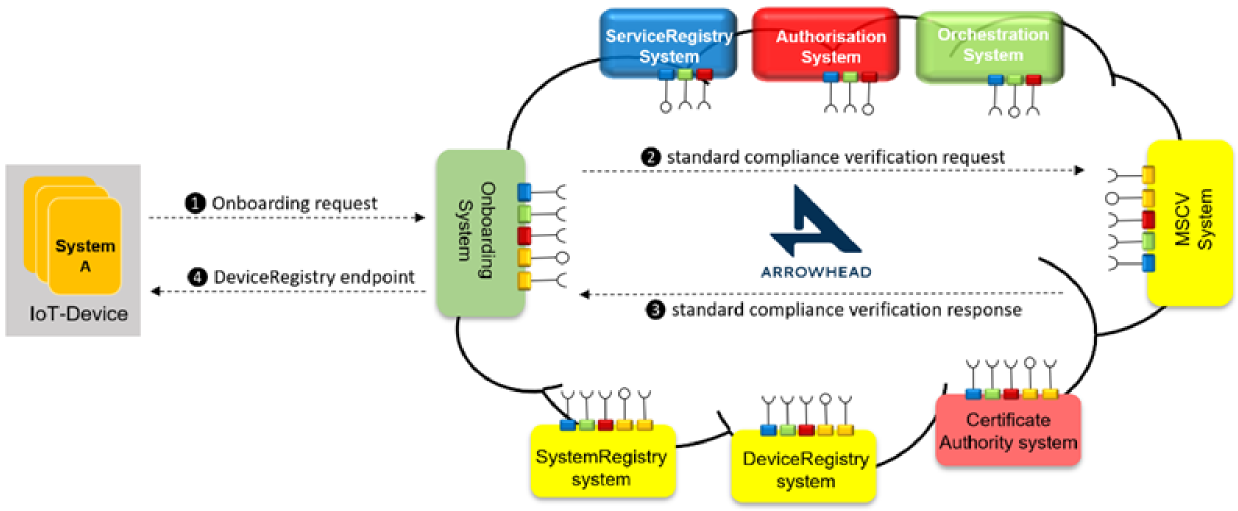


Figure 6.2 MSCV system integrated in Arrowhead local cloud

6.4 Status of Work

A first prototype of the MSCV system is already available in Arrowhead GitHub.
Currently, we are working to integrate it with WP9 use case “Linking building simulation to a physical building in real-time” and to provide new features such as, standard compliance verification in run-time.

7 Vital-IoT

Vital-IoT is a smart city platform that enables the integration, orchestration and visualization of IoT services’ data streams from multiple systems. It provides a set of data models and interfaces enabling collection and annotation of information from diverse systems in a simple developer-friendly format and in a way that ensures their unified and interoperable representation. It also provides a management environment, which permits unified supervision of diverse IoT device systems and data streams from a single entry point.

7.1 Rationale for Integration

In the primary version of Vital-IoT, to allow heterogeneous systems to be connected, a set of specifications called PlatformProvider Interface (PPI) were used, which defined the interface between the Vital-IoT platform and third-party IoT systems. An implementation of the PPI is all that was required by IoT platform providers who wish to make their systems compliant with Vital-IoT. PPIs could be also used to provide access to individual (IoT-related) data sources and datasets.

The PPI has been defined as a set of RESTful web services that are marked as either mandatory or optional. The classification of the PPI services into optional and mandatory minimizes the effort required from IoT systems providers to integrate their systems into Vital-IoT while offering a way of exposing information that may be needed in specific scenarios. The PPI is registered to the Vital-IoT platform to allow it to retrieve:

- Information about the IoT systems (e.g. their status)
- Information about the services that an IoT system exposes (e.g. how to access them)
- Information about the sensors that an IoT system manages (e.g. what they observe)
- Observations made by the sensors that an IoT system manages

Although PPI facilitates the integration of new services, the process of adding new services still was a very complicated operation that was not achievable without the intervention of Vital-IoT admins and a set of manual installation and configurations.

However, using Arrowhead framework and its services, it is possible to discard many steps of the manual integration of new services and facilitate the process of adding them considerably. Considering the mentioned advantages, we have completely discarded the PPI service form Vital-IoT, and inserted an Arrowhead interfacing module in Vital-IoT's dashboard to communicate with Arrowhead platform services to obtain information regarding required services and measurement systems conveniently. Using this information, it is possible to communicate with the provider system without any considerable complexity.

7.2 Technical Details

Vital-IoT is composed of the functional modules described in Figure 7.1.

The User Application of the Vital-IoT dashboard is made of three main modules. The first one is the "User Authentication" module, composed of the user registration page and user login page. The second one, the "Arrowhead Interfacing" module, is dedicated to communicating with the Arrowhead orchestration service in order to provide the end-user with all of the services he is looking for. And finally, the "Data Visualization" module, which shows the measurement data provided by the sensors in the requested time window.

In the back-end part, we have a Flask application providing services for user authentication, user registration, service configuration management and providing data for the visualization. Moreover, the Flask module offers resources to calculate statistical measures such as mean, minimum and maximum measured values.

Another essential part is the "Data Collector". Mostly, it acts as a listener, subscribing to the MQTT topics and collecting the upcoming data. After that, such values are passed to a data validator to ensure compatibility with Vital-IoT's data format, in order to prevent errors in the data visualization phase. In the end, the data is saved in the MongoDB database.

We set up a Mosquitto broker service, responsible for gathering data from all of the data providers, without having them to configure their own broker service. Nevertheless, Vital-IoT is also capable of collecting data from other external MQTT brokers.

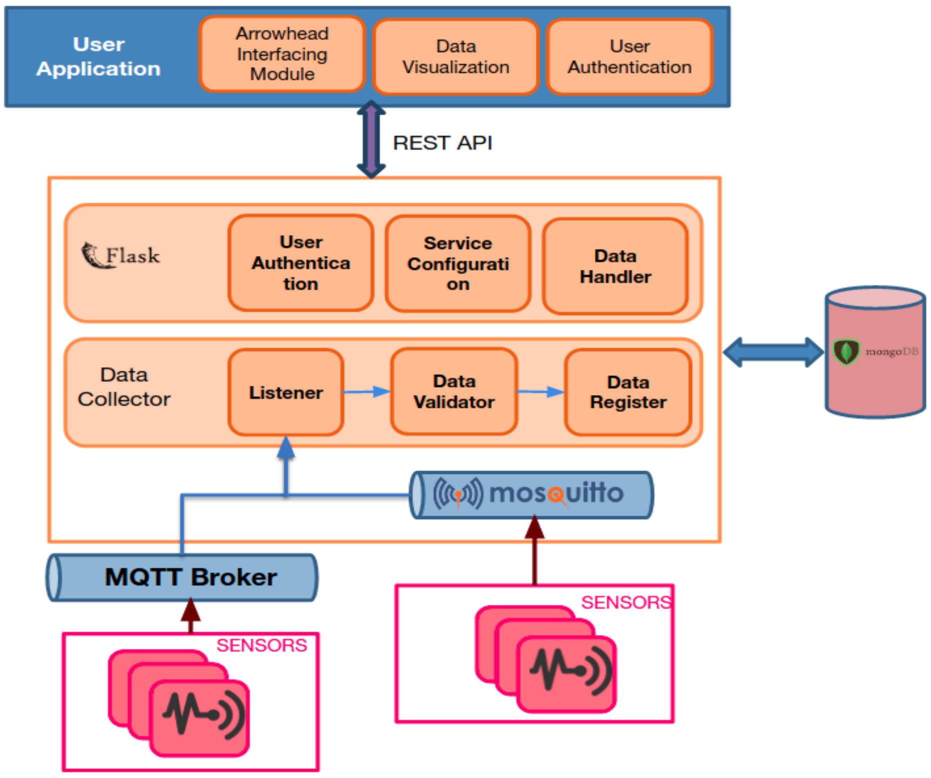


Figure 7.1 Vital-IoT functional topology

7.3 Integration in Arrowhead Framework

In order to make the Vital-IoT system compatible with Arrowhead Tools, it was necessary to perform some modifications in the architecture and functionalities of our Smart City platform. Figure 7.2 shows the implemented modification on the Vital-IoT platform.

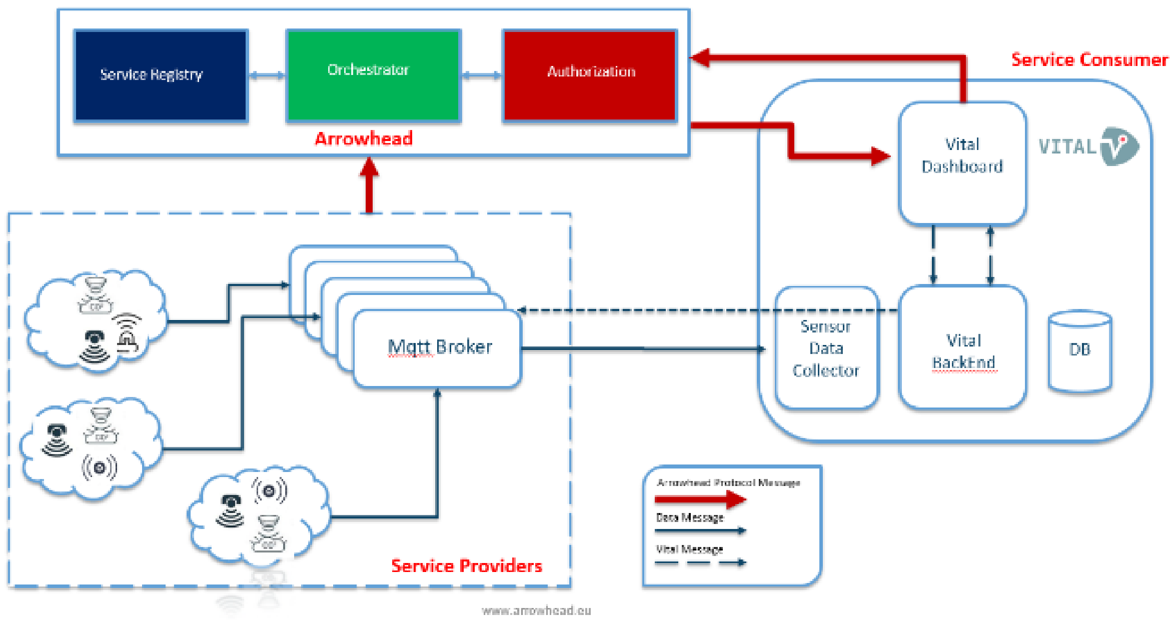


Figure 7.2 Vital-IoT integration with the Arrowhead Architecture

First of all, the measurement service (which in our case are the sensors), must be registered in the list of available providers. This is achieved through a POST request using Arrowhead's service registry, using the following URL:
http://{registeration_service_address}/serviceregistry/register

Furthermore, using the mentioned POST method, the service provider will also be registered in the list of providers if it is not registered before. However, at this stage of the process, besides the usual main information for registering a provider, it is necessary to include information which is crucial for initializing the communication and collecting information from the sensors (such as the broker address, the broker port, the topic, etc.).

On the other hand, we may have different sensors, which probably use diverse data structures to convey their information. Thus, it is necessary to have a common understanding of the transferred data between the provider and the consumer. To achieve this, we have suggested creating a dictionary to declare the necessary instructions for having a common understanding between the provider and the consumer. Moreover, such information is critical for the visualization purpose. e.g, which keys refer to timestamp, measured values, measurement units, etc.

Looking at the "metadata" field, here is where we have decided to insert the information regarding the connection properties of the services and necessary information for the visualization purpose.

In the "providerSystem" field, the main key is the "systemName" which in our concept should be equal to a group of sensors that we consider a system.

The other important field is "serviceDefinition" that is the key element for the orchestration phase and discovering the services. As we mentioned previously, it should follow the form of "vital_[measuredValue]_sensor" to assure consistency with the Vital-IoT and for comparison purposes while having data from various systems e.g. temperature in different cities.

Likewise, we need to register our consumer system that is the Vital-IoT, using the following URL:
http://{registeration_service_address}/serviceregistry/register

This step is mandatory since the information of the registered consumer must be used for the authorization process. In this phase, we specify which consumer can access the information of which providers and services. In our case, we authorize the Vital-IoT system to access the measurement services available from our sensors. This is done using the following URL:
http://{registeration_service_address}/authorization/mgmt/intracloud

It is important to make sure that the "consumerSystem", the "providerSystem" and the "serviceDefinition" are the same systems we have specified in our message body.

The Vital-IoT starts the orchestration process using a POST request and by specifying what kind of service it is looking for, using the following URL:
http://{orchestration_service_address}/orchestrator/orchestration

This leads to receiving the response message with the list of available requested services. The user selects the services he is looking for and saves the configuration information.

Following this action, the configuration information (the whole data received from the orchestration service) is sent to the backend and saved in the configuration collection alongside with the configuration data of the other service providers. In order to achieve this, a POST request must be submitted to the backend with the following URL:
http://{vital_iot_service_address}/config-update

Meanwhile, the backend starts the process of subscription to the broker with the topic mentioned in the configuration received from the frontend.

The flow of the data received from the broker is saved in the database using the dictionary provided in the system registry phase and the adapted data will be sent to the frontend to be visualized when requested.

7.4 Status of Work

Available Features	Present Gaps
- Establishment of direct communication between Arrowhead Framework and Vital-IoT dashboard - Integration and visualization of data from diverse sensor types and sources. - Unified data storage solution with an interface towards run-time systems - Integration of Edge Computing and pre-processing of data collected from	- Scalability and remote installation and control of the Edge Computing side of the architecture - Improvements of data visualization - Dashboard personalization based on user preferences

different devices and sensors - Configuration and measurement data validation - Providing statistical data measurements and comparison of values where possible - Presence of internal MQTT broker to facilitate data transfer of external measurement devices	

8 IKERLAN Tool Adapter

8.1 Rationale for Integration

Arrowhead standard architecture is shown in Figure 8.1. To run this architecture all involved tools (Tool1, Tool2, Tool3) must call Arrowhead framework for registering and getting IP addresses and Ports of other tools. This implies that (1) in every tool, the program code must include the logic of this management and in addition to this, (2) each tool may be programmed in a different programming language, so the same logic must be programmed several times in different languages.

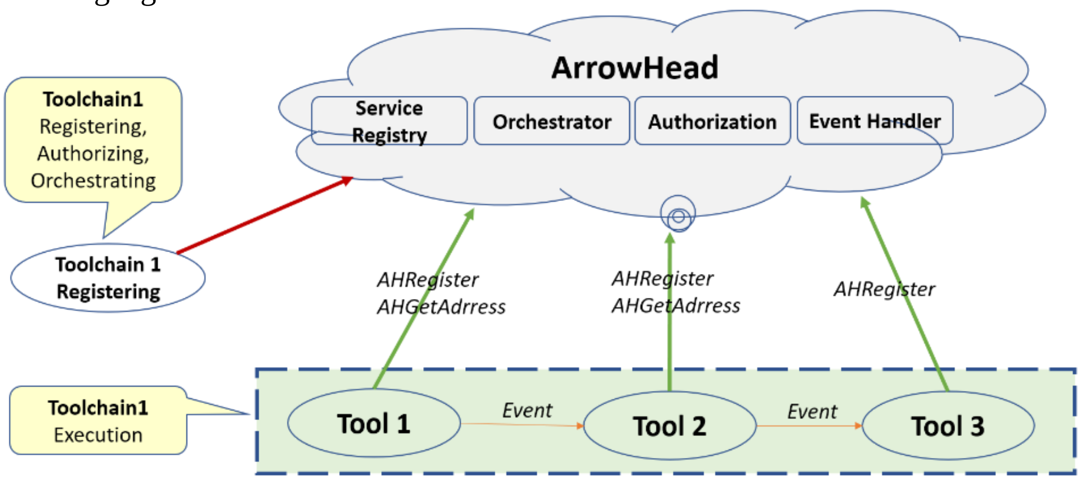


Figure 8.1 AH standard architecture

As requested from use cases of Fagor Automation and Fagor Arrasate, and with the goal of reaching to a single solution that (1) is independent from the technology in which the tool is developed and (2) hides to the tool some of the complexity of the interaction with Arrowhead framework, a middleware layer shown in Figure 8.2 is proposed.

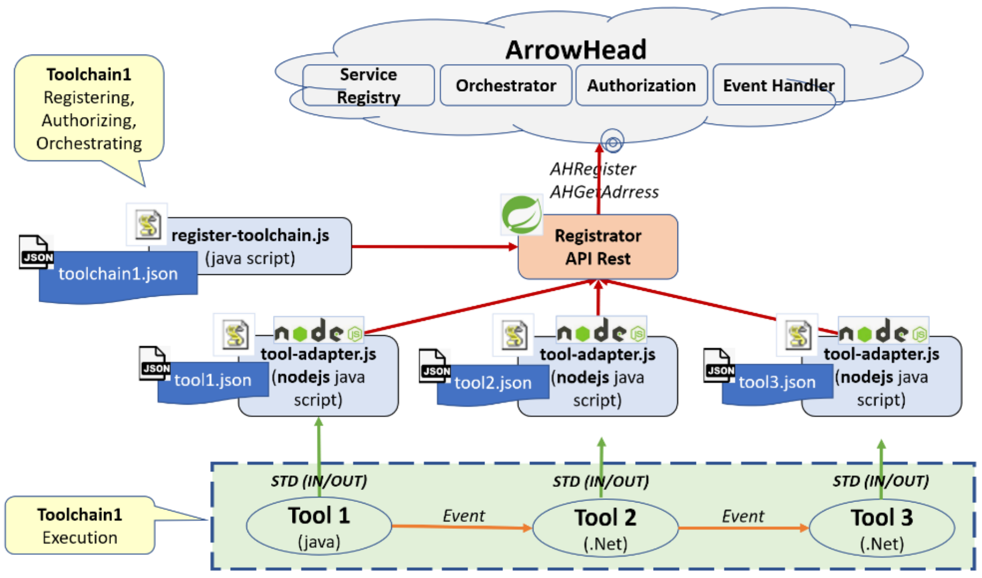


Figure 8.2: middleware layer

Following sections explain the different components of the middleware layer in detail.

8.2 Technical Details

The middleware is composed by Software Components and Configuration Files.

8.2.1 Software components

1. **Registrator API Rest**. This is a Java REST WebService (developed with Spring https://spring.io/) implementing the following services:
   1. `registerConnections:` This service register in AH framework a toolchain that is passed as parameter.
   2. `updtateToolAddress:` This service updates in AH framework the IP and Port of a tool name that is passed as parameter.
   3. **`getUriOfService:`** Gets the uri of a service name that is passed as parameter.
   4. **`getEventPublishService`**: Return the AH event publishing service address.

2. **Nodejs** (www.nodejs.org). It is a JavaScript execution environment oriented to asynchronous event management in a scalable network.
3. **register-toolchain.js.** This JavaScript file is executed in **Nodejs** environment and calls the **Registrator API Rest** to register and authorize the *toolchain* that is passed as parameter as a configuration file. See section *Example of toolchain configuration file* for an example.
4. **tool-adapter.js.** This JavaScript file is executed in **Nodejs** environment. Each *tool* calls the tool-adapter passing its IP and Port in a configuration file that is passed as parameters. See section *Example of tool configuration file*, for an example.

8.2.2 Configuration Files

This section explains configuration files that are passed as parameter to the Software Components described above.

8.2.2.1 Example of **toolchain** configuration file

This section shows a JSON configuration file corresponding to a ***toolchain*** called *thermostat*. Main parameters inside the file are the *from-to* connections that define the information flow.

thermostat.json
```
{
  "toolchain": "thermostat",
  "connections": [{ "from": "temperaturesensor", "to": "heater",     "channel": "temp2heat" }]
}
```
Note that registered tools have no information about IP Address and Port. This data will be defined later (each tool will call the **tool-adapter.js** java script with these parameters).

8.2.2.2 Example of **tool** configuration file

This section shows a JSON configuration file corresponding to a ***tool*** called *heater*. Main parameters inside the file are the registrator url (basically IP and Port) and the listening port of the tool.

heater.json
```
{
  "tool": "heater",
  "registrator_url": "http://localhost:9000",
  "listen_iface": "Ethernet",
  "listen_port": 55116,
  "spawn": {
     "command": "node",
"args":      ["-e",                              "require('readline').createInterface(process.stdin).on('line',l=>{const
obj=JSON.parse(l);console.error(obj.payload>28?'OFF':'ON');});setInterval(()=>console.log(JSON.stringify({
metaData:{type:'request',to:'temperaturesensor'},payload:''})), 1000)"],
     "options": {}
         },
  "toolAddressCacheTime":10,
  "launchOnConnection": false,
  "relaunchIfExit": false
}
```

8.3 Integration in Arrowhead Framework

This section shows how all the software components and configuration files describe above fit together. A typical process is as follows:

1. The JavasScript **register-toolchain.js** file is executed in ***Nodejs*** with a parameter consisting of a configuration file defining the toolchain. For example the file **thermostat.json** defines a toolchain consisting of a temperature sensor and a heater in which the temperature sensor (from) sends information to the heater (to) via AH Event Handler:

   thermostat.json
   ```
   {
     "toolchain": "thermostat",
     "connections": [{ "from": "temperaturesensor", "to": "heater",     "channel": "temp2heat" }]
   }
   ```
   This JavaScripts file **register-toolchain.js** invokes **Registrator REST Web Service::registerConnections** service that will:
   - Register temperaturesensor and heater in **AH Service Registry**.

- Authorize temperaturesensor to access heater in **AH Authorization**.
- Registers communication events in **AH Event Handler**.

2. The JavaScript file **tool-adapter.js** is executed in **Nodejs** for each tool defined in the toolchain, in this case:
   a. **tool-adapter** is invoked with configuration file **temperaturesensor.json**
   b. **tool-adapter** is invoked with configuration file **heater.json**

   The JavaScript **tool-adapter.js** invokes **registrator REST Web Service::updateToolAddress** with the configuration file and will:
   - Update the IP and Port of the tool in the **AH ServiceRegistry**
   - Get the IP and Port of the **AH Event Handler Service**
   - Orchestrate temperaturesensor and heater in **AH Orchestrator**.

8.4 Status of Work

Available Features	**Present Gaps**
- Software Components and Configuration Files of middleware developed and tested. - Integration with AHT Platform in Ikerlan tested and working. - Middleware ready for using in Fagor Automation and Fagor Arrasate test cases.	- Detailed documentation of Software Components usage pending. - Detailed documentation of Configuration Files pending.

9 Extended Historian Service

9.1 Rationale for Integration

An Extended Historian Service (EHS) is developed as an optional Arrowhead core service. It is an extension of an already planned core service Historian Service (or meanwhile renamed Data Manager). The intention is to provide a broader functionality than the current concepts provide, which better supports data analytics scenarios. The EHS will be used as base for prototypical implementations of those tools in scope of the Arrowhead Tools project, which are evaluated in use case task T9.4 “Production support, energy efficiency, task management, data analytics and smart maintenance”.

Production systems today are most likely controlled by PLC-like systems based either on special hardware or on industrial ready PC hardware. The controller is interfaced to necessary sensors and actuators. Those systems have the small but important scope to guarantee the main functionalities of the production systems.

Often there is the requirement to optimize the production process long-term regarding performance, product quality or reliability of equipment. In order to use the potential, the data of products and the process information have to be analyzed. The control procedures can then be re-defined based on the results of the data analysis.

Data acquisition systems and data analytics functionalities are necessary for that purpose. Those data acquisition systems access the data most likely from the controller, since most sensor values have to be considered in the control procedure. But in some cases, additional sensors are connected directly to the data acquisition system.
Most data acquisition systems come with some kind of data analytics functionality. But in some cases there is a need to connect external data analytics software to the data pool. Then there is a need to access the interfaces of the data acquisition system by external data analytics software, which can become a difficult work and if it becomes too difficult, then company-internal research work can be blocked due to the big efforts.
From the economic perspective of an industrial company, the data acquisition and data analytics system are separate systems beside the control system, which requires separate investment for hardware, licenses, commissioning and maintenance of the system necessary. This is expensive and sometimes the reason, why appearing data is not exploited as good as it could be.

9.1.1.1 Data acquisition and analytics by use of the Arrowhead Framework

Delsing et al. [7] describe the Historian system of the Arrowhead Framework as usable for storage, processing and visualization of data created by services of a local cloud. It should provide a wide range of protocols and data models. The historian system can be orchestrated to consume any services in the local cloud. The historian system consists of several services:

- Historian service: provides an operation to store sensor data (PutData).
- FileSys service: enables clients to store and delete files and folders like in a distributed file system.
- Filter service: enables clients to retrieve data stored by the Historian service.
- Service information: provides information about the encoding of files, which is primarily based on SenML, but has different basic encodings like JSON, CBOR or XML.
- Service meta-data: provides information e.g. about active devices.

9.1.1.2 Motivation for the Extended Historian service

The initial idea and motivation of the Historian system of the Arrowhead Framework was really future oriented and still holds: **Provide a data acquisition and analysis system directly with the communication platform.**
However, there are several weaknesses regarding the current design and implementation of the Historian system, which should be overcome with the implementation of the EHS:

- The Historian service is described as passive only. It implements only a PutData operation. But in real scenarios it sometimes is necessary to acquire data sets synchronously and at defined sample times. Thus, the service should include a kind of job management to gather the data like other commercial data acquisition systems do.
- The use of file based data is potentially very slow for frequent data samples from a large amount of sensors. It should be replaced by fast and reliable Open Source database management systems.
- There are a lot of high performance data analytics systems on the market and even there exist a lot of Open Source solutions. Thus, the Extended Historian service should make

it possible to interface to those systems, instead of providing the data encoded as SenML files.

- There is a need for input of heterogeneous sensor data. Thus, the service needs a plug-in concept for adapters to the data sources.

9.2 Technical Details

Figure 9.1 provides an overview of the Extended Historian service (EHS) and its application context. Therein, the EHS consists of the following component types:

- Historian Database: the component, which stores the incoming data.
- Job Scheduler: the component, which is able to trigger data source adapters to gather data from sensors and to put them into the Historian Database.
- UMAA-System: a component providing functionalities for role based user management, authentication and authorization (UMAA).

The EHS is accompanied by several applications and adapters, which organize data exchange with non-Arrowhead data sources or sinks:

- EHS Configuration System: this is a graphical user-frontend for configuration of the internal data model and the job-scheduler of the EHS as well as for configuration of the data source and sink adapters. It will be developed in WP5 of the AH Tools project. The EHS Configuration System can be designed as internal module of the EHS.
- Data Source Adapter: a component, which connects to a sensor or other data source. It is able to send and receive data over the industrial communication system used by the sensor. Figure 9.1 depicts the following possibilities for initiating data acquisition:
    - The EHS job-scheduler triggers the data source adapter to get data from the sensors and to push it to the Historian Database.
    - The data source adapter triggers the data acquisition from the sensor and pushs the measured values to the Historian Database.
    - The sensor actively triggers the data source adapter to take over the data from the sensor (push) and to put it into the Historian Database.
    - The orchestrator triggers the sensor to push data to the data source adapter and to put it into the Historian Database.
- Data Sink Adapter: a component, which connects the historian database to the data analytics software. It is able to provide the data in a data format and over a network connection, which is usable for the analytics software.

Those EHS companion applications are separate processes. They will use the gRPC technology for interprocess-communication with the EHS core system.

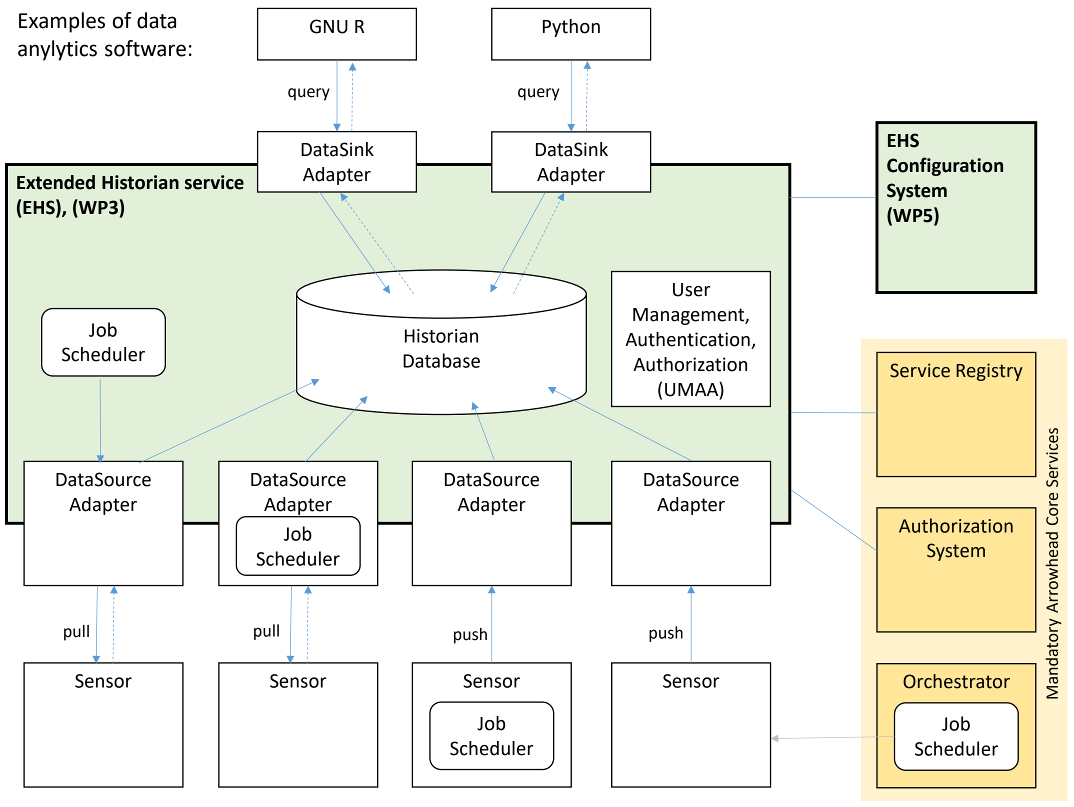


Figure 9.1: Extended Historian Service (EHS) and its application context

9.3 Integration in Arrowhead Framework

There are several Arrowhead Framework core services around the EHS:

- Service Registry: the service registry is used to make the EHS known in the Arrowhead local cloud.
- Authorization System: the authorization system is used to validate access to the EHS by sensors (push scenario) and to get access of the EHS to the sensors (pull scenarios) in cases, where the sensors are themselves Arrowhead application services.
- Orchestration System: the orchestration system may initiate data transfer from sensors to the EHS in cases, where the sensors are themselves Arrowhead application services.

9.4 Status of Work

We have currently following status regarding the EHS:

- There is a specification for the EHS available including a use case description, description of technology candidates and a definition of basic building blocks of the software architecture.
- The main technologies (Apache Spring Boot, Quartz Scheduler, gRPC for Java) have been tested and integrated including tests of parallel working program threads.
- The EHS base system is under development, with a connection to a PostgreSQL database for storing the time series.

10 WAE (Web-of-Things Arrowhead Enabler)

The WAE is a tool that enables the inclusion of a Web of Things ecosystem into an Arrowhead Local cloud, by creating one service for each Web Thing. In detail, this is a an essential middleware component that acts as a discovery bridge for the WoT layer.

10.1 Rationale for Integration

In order to present correctly the tool and its integration, it is necessary to introduce briefly the Web of Things (WoT). The chaotic world of the Internet of Things is characterized by tens of different technologies, protocols, and architectures for interconnecting Smart Things to the Internet. Because of its fragmented nature, one of the biggest challenge of the IoT landscape is constituted by the lack of interoperability. For this reason, starting from 2015, several universities and companies in the WoT ecosystem seamlessly joined the W3C working group for the definition of a Web of Things (WoT) standard, whose goal is claimed to counter the fragmentation of the IoT, by defining a reference architecture, the communication patterns and the interfaces of the Things; the rationale is to enable the interoperability among IoT systems, regardless of the underlying stack implementation and of the networking technologies being used.

The Web of Things paradigm presents nowadays the challenge of implementing a discovery operation of Web Things that supports matching and lookup as well as orchestration. In fact, Thing Discovery has been repeatedly claimed as a desirable feature to cope with several problems, for instance, the mobility of Things.

10.2 Technical Details

We here briefly describe the implementation of the main components of our architecture released in D4.2 (sub-document O2). The Service Registry is a JAVA server that exposes some REST APIs. In particular, we used the API already available as open source project. All Web Things involved in the scenario have been implemented and instantiated by using node-wot, the official W3C framework for the WoT. The WAE component has been designed as a Web Thing - for being able to natively speak to other W3C WoT entities - and as an HTTP client - in order to use the Service Registry's APIs. As shown in the table below, following the paradigm **Properties, Action, Events** as explained in the official WoT documentation, the WAE Web Thing exposes the listOfWebThings Property for listing all the already known Web Things it has published. Additionally, it exposes also the startCrawling and the query actions. The first is automatically invoked once the WAE has been deployed to look for new Web Things that have been published on the Thing Directory. The second one is invoked by a WoT Consumer in order to query the SR and to get the list of services that match its request. Finally, a generic event newWebThing is fired each time new Web Things have been discovered by the WAE. Both the HTTP client of WAE and the AH Consumer have been customized for our need by taking advantage of the already existing open source NodeJS Arrowhead Client. Each WoT Consumer is a Mashup Application, i.e., a javascript application that uses node-wot framework as a library and simply consumes multiple Things to interact with them in order to collect data and manipulate it for its needs. Lastly, the ATM is an ExpressJS web server that maps each Web Thing Affordance to a REST API and that uses the node-wot as a library behind the scenes in order to interact with the Web Thing it represents.

Name	**Type**	**Description**
listOfWebThings	Property	List of all the Web Things the WAE is aware of.

startCrawling	Action	Start to look for new Web Things that are published on the TD.
query	Action	Forward the query of a Wot Consumer to the WAE.
newWebThing	Event	This event is fired when a new Web Thing has been discovered on the TD by the WAE.

10.3 Integration in Arrowhead Framework

In order to support different types of external consumers interacting with the WoT ecosystem, we propose the layered architecture released in D4.2 (sub-document O2). The architecture consists of three conceptual layers: the Physical layer, the WoT Layer and the Arrowhead Layer. Entities on each layer can communicate directly with other entities belonging to the same layer as they are assumed to use the same application protocol.

Each sensor gets abstracted onto a Web Thing, according to the WoT paradigm. Each Web Thing is then registered onto a special Thing Directory, a standard registry for the WoT ecosystem. The central component of the WoT Layer, the WAE, can be classified as a WoT Mashup Application. In detail, it periodically queries the Thing Directory to detect new Web Things right after they spawn (i.e. the binding with the actual sensor is generated). As new Web Things are detected, the WAE performs a publish operation for each of them against the Service Registry in the Arrowhead Layer to publish Web Things as new Arrowhead services. The communication between the WAE and the Service Registry is the sole case of inter-layer communication channel, in which a component (in this case the WAE) acts as a proxy able to use two different communication protocols.

Furthermore, each Web Thing is extended onto the Arrowhead Layer by a new module, called Arrowhead Thing Mirror (ATM). The ATM exposes the Web Thing service endpoint as an HTTP Web Service in the Arrowhead local cloud. Note that a Web Thing and its relative ATM can run on the same piece of software as well as in separate components connected by a custom communication link. The ATM plays, to some extent, the role of an Arrowhead service adapter, however, it does not perform publish operation, as they are handled by the WAE.

The record published by the WAE exposes by default the endpoint and the metadata of the ATM related to the Thing, while the JSON-LD description of the Thing at the WoT Layer is converted to a string and encapsulated in the newly created ``TD" subfield of the ``metadata" JSON field of the service record. This way, a consumer can interact with the Web Thing in two ways, depending on its communication capabilities:

1. An HTTP-enabled Consumer queries the Service Registry, selects the service that provides the type of data needed and gets the endpoint of the service, which corresponds to the endpoint of the related ATM. The Consumer then performs the consume calls against the service offered by the ATM which, in turn, queries the Web Thing and retrieves the data point. Response data travels then backwards to the Consumer.

2. A WoT-enabled Consumer queries the WAE, which retrieves the list services from the Service Registry. As the consumer is only able to interact with WoT-enabled systems, the WAE decapsulates the related TD from the field ``metadata" of the service record and sends it back to the consumer. The latter is then able to select a Web Thing among

the received ones and query it directly; the Web Thing endpoint is enclosed in the TD itself, thus the actual endpoint, which belongs to the ATM, is ignored.

The whole interaction for the two types of consumers is shown in detail through the sequence diagram in Figure 1. In particular, it shows mainly two patterns:

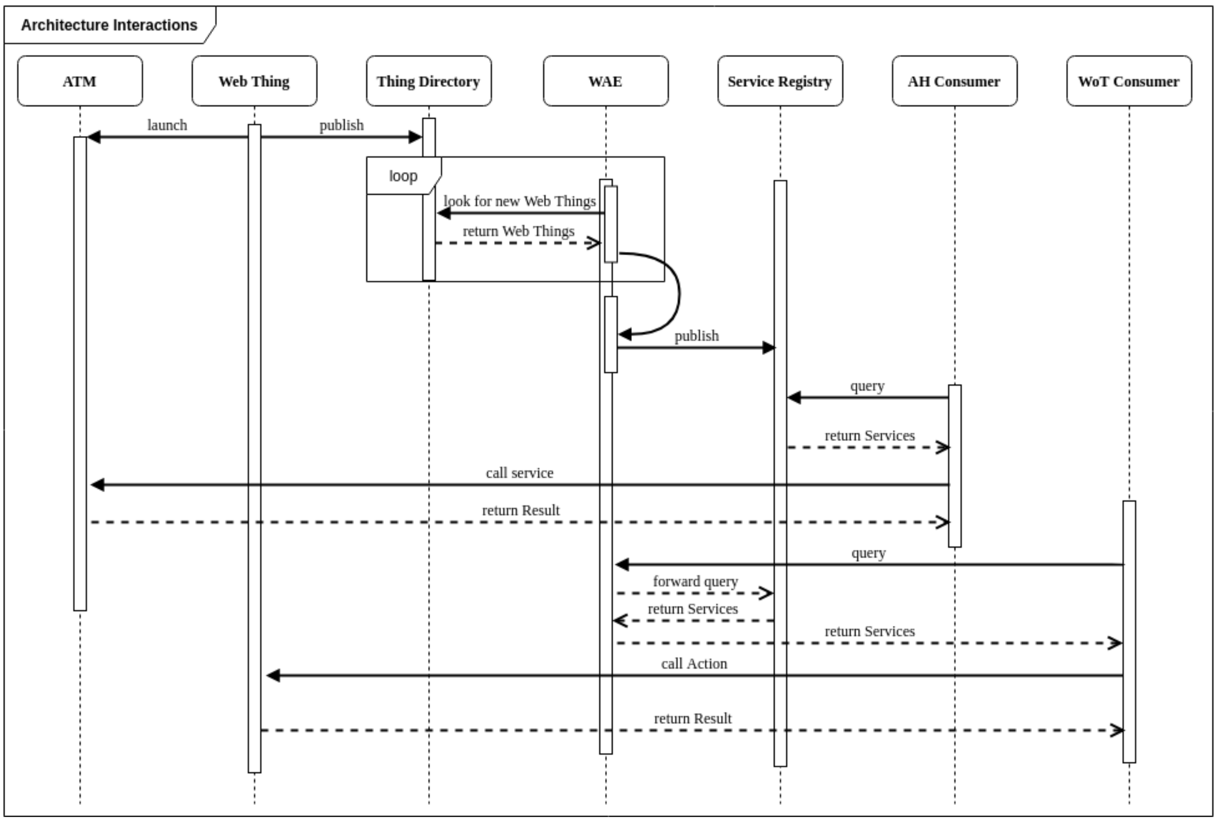


Figure 10.1 Sequence diagram presenting the interactions of all the components of our three-layer architecture

Web Things' publication on SR when a Web Thing is generated, it automatically instantiates an ATM to be able to fulfill a request coming from an AH Consumer. At the same time, the Web Thing publishes itself on the Thing Directory, according to the draft of the W3C standard proposal. The WAE is in charge of keep checking if new Web Things appear on the Thing Directory, and in case to publish themselves on the SR. This can be achieved in two ways, depending on the WAE implementation: either the WAE polls the Thing Directory or the WAE is implemented as a Web Thing, so it can subscribe to Thing Directory's events. The SR is listening from queries of services' consumers and replies with all the Services is aware of, including the Arrowhead ones that however are not shown in this diagram.
Web Things' consuming once the services related to Web Things become available on the SR, they can be queried by consumers. Depending on the kind of consumer, there are different interactions. The easiest case is the one involving an AH Consumer, since it has only to query the SR, to get the list of services and to directly interact with them through their ATM. A Wot Consumer instead requires more steps in order to be able to retrieve the Thing Description of the Web Thing that should be consumed. First, it has to send the query to the WAE, since it might not be able to speak directly to the SR. Hence, the WAE forwards the request to the SR

and waits for the list of services that match the query originally coming from the WoT Consumer. Finally, WAE returns the list to the WoT Consumer, that is now able to consume the Web Thing and to interact with it, for instance by invoking one of its actions.

10.4 Status of Work

Available Features	**Present Gaps**
- Discovery of Web Things - Add new web things seamlessly to the local cloud - Interact with them through a WoT consumer - Interact with them through an Arrowhead Consumer	- No interaction with Authorisation - No interaction with orchestrator - Not possible the other way around (include an arrowhead local cloud into a WoT ecosystem) All these are planned

11 Python Client Library

This is a library that supports the creation of Arrowhead systems and services written in the Python programming language.

11.1 Rationale for Integration

Python has seen a steady rise in popularity over the last several years, especially in the fields of AI and machine learning. This library serves as a bridge to those and other communities that want to integrate their Python programs in an Arrowhead local cloud.
The library has two main goals:

- Make it easy to create your own services.
- Automate the registration of services.
- Use a standard interface to use core services.

11.2 Technical Details

The current version of the library supports Python 3.8 and uses the Flask, Requests, and Gevent 3rd party libraries.

11.3 Integration in Arrowhead Framework

The library currently supports the service registry, orchestrator, and authorization core systems with support for some, but not all, of their core services. Support for other core services will be added as needed.
Regarding security, the library only supports the CERTIFICATE level security at this point, but work on the INSECURE level is being developed.

11.4 Status of Work

Current version is available on PYPI (https://pypi.org/project/arrowhead-client/).
Further developments are undergoing but not yet published. However, they can be found in the Github repository (https://github.com/arrowhead-f/client-library-python).

12 Semantics translator

The need to establish dynamic and operational interoperability at the system of systems level is challenging in heterogeneous semantic environments with a variety of data formats and models, legacy systems and semantic spaces defined by different ontoligies and standards.
System-level ontologies capture detailed characteristics of contexts, models, and interactions. Thus, the semantic space is complex and it is also expected to be non-static due to reconfigurations, updates, technology migration, maintenance etc [38].
The Semantics translator is a concept under development that takes advantage of machine learning methods to translate messages sent between services, thereby aiming to signficantly reduce the system of systems engineering effort required and improve the scalability of the Arrowhead Framework beyond the limits of manual system design and orchestration.

12.1 Rationale for Integration

A machine-learning concept for semantic translation is investigated as a means to enable scalable dynamic and operational interoperability in heterogeneous semantic environments. Aiming to lower the cost of system integration and system of systems engineering, we are investigating techniques for the translation of messages exchanged by Arrowhead services where data, metadata, goals and policies can all be incorporated in the process.
This approach is essential for scalable translation, because the meaning of exchanged messages depend on the intended use of the information (descriptions of meaning are relative). A pragmatic approach towards scalable dynamic interoperability is a machine learning approach where metadata and the purpose and policies of the system of systems are used for message-driven generation of the translators. This is how machine learning models are successfully used to correctly interpret complex inputs like the color components of pixels in deep learning based computer vision models, or characters in models for natural language processing. Thus, in principle the approach investigated is feasible and is the only approach identified that can lead to cost-efficient and scalable solutions in the context outlined above.

12.2 Technical Details

A machine learning based concept for semantic translation that can enable dynamic and operational interoperability at the system of systems level is outlined in [39] and is partially implemented and evaluated in a simulated environment in [40], where a translation accuracy of 75% is achieved using an encoder-decoder type deep learning architecture as illustrated in the figure below.

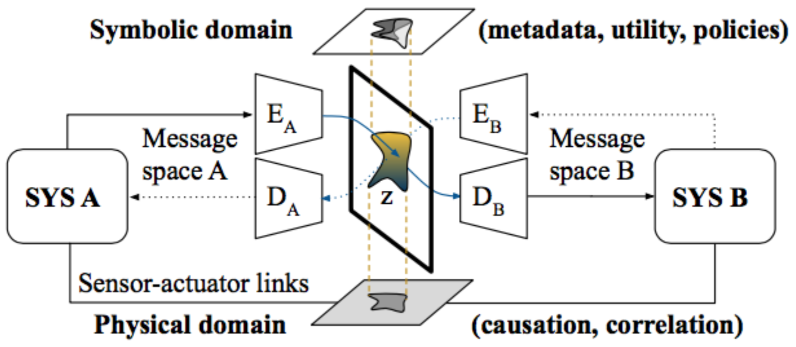


Figure 12.1 Encoder-decoder concept for message translation investigated and tested in a simulated environment in the paper [40]. Deep learning based encoders, $E_{A,B}$, and decoders,

$D_{A,B}$, are trained with unsupervised learning to translate messages between two incompatible heating and ventilation systems, SYS A/B, via an intermediate embedding z.

12.3 Integration in Arrowhead Framework

The Semantic translator is a new concept under development [39]. Partners have performed tests in a simulated environment that show promising results [40] and an experiment with a lab-scale production process is ongoing. The method needs to be further developed to meet the requirements of a production environment. Further development and testing is thus required before integrating the machine learning concept for semantic translation in the Arrowhead Framework.

12.4 Status of Work

The work so far has focused on establishing an unsupervised deep-learning based architecture for service message translation [40] and further work is ongoing to implement additional aspects of the full concept outlined in [39]. In particular, we are investigating how to incorporate additional metadata in the translation model like RDFs, and more user-friendly utility and policy descriptions, thereby aiming for Arrowhead Framework compliance.

13 Code generation

13.1 Rationale for Integration

Reducing the need for engineering effort in updating code for a system that shall consume a specified and produced service.

13.2 Technical Details

This work contributes with an engineering approach for creating interoperability among heterogeneous systems in service-oriented architecture (SOA) environments by generating an autonomous consumer interface code at run time. The proposed system makes use of service interface descriptions to dynamically instantiate a new autonomously generated interface that resolves communication mismatches between provider and consumer. As illustrated in the following figure, the generated interface provides service contract translation to overcome interoperability issues.

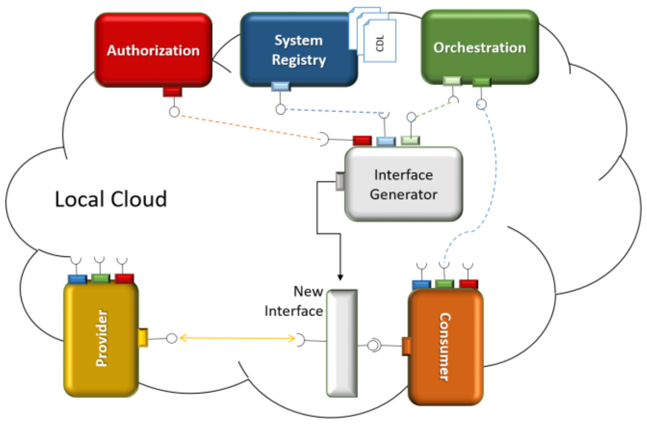


Figure 13.1 Arrowhead consumer interface generation scenario.

Furthermore, the work includes an analysis of service contracts and the capability of machines to determine interoperability mismatches. A comparison have been made of three commonly used description languages, WSDL, WADL, and OpenAPI. They have been analyzed and compared based on key requirements for the development of new tools such as interface generators and service contract translators. The results of the comparison highlight the language limitations to fully describe the service contract. The misalignment between the available interface description languages and the service contract requirements is outlined to help extend these languages to fully describe the service contract in future improvements.

13.3 Integration in Arrowhead Framework

Currently at prototype status

13.4 Status of Work

Working code and two papers describing core ideas behind. The current GAP is a well defined approach and methodology for identifying the miss-match between the service contract of the producer and the consumer.

14 Integration of OPC-UA server with late binding

An Arrowhead Framework system that intercede for an OPC-UA server has been developed. To demonstrate its potential, a Siemens PLC with an integrated OPC-UA server is connected to a model assembly with two machining stations and four conveyor belts. OPC-UA promotes a service oriented architecture, where one can at runtime discover the "nodes" or capabilities of the asset (in our demonstrator this is the complete assembly line). These capabilities are then discovered by the Arrowhead Framework and registered as services with the service registry.
Emerging behavior includes finer granularity of cybersecurity and late binding. Authorization is at the service level rather than at the system level. That is some systems are allowed to consume only certain services and not others. With OPC-UA, which uses the same certificate scheme, once access is granted, one has full access to all nodes. In our demonstrator, we show that accessing a service can be done at run time by describing the desired service.

14.1 Rationale for Integration

OPC-UA is already used in industry and the Industry 4.0 Platform often refers to it as a standard. Since the Arrowhead Framework interoperates with different IoT and CPS solutions, interacting with OPC-UA devices is natural.
Any investment done with OPC-UA solutions (i.e., brown field) can easily adopt the Arrowhead Framework and that is what is shown here.

14.2 Technical Details

The Arrowhead Framework application use Eclipse Milo to enable it to be an OPC-UA client and scan the available nodes.
Subscription to events is also possible.

14.3 Integration in Arrowhead Framework

The integration with the Arrowhead Framework is smooth and complete. All that is needed is the URL of the OPC-UA and its main node.

14.4 Status of Work

The Arrowhead Framework OPC-UA is functional and has been logging status states of an assembly line in northern Sweden with the millisecond resolution to a Jotne local cloud in France. Any changes in capabilities of the assembly line/PLC is automatically handled by late binding.

15 Translator system

15.1 Rationale for Integration

The Translator system is responsible for translating between different communication and semantic protocols.

15.2 Technical Details

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.

15.3 Integration in Arrowhead Framework

Part of Arrowhead Framework.

15.4 Status of Work

The system is updated and is now targeted for the next official release of Arrowhead.

16 Exchange Negotiation Service (from Productive 4.0)

16.1 Rationale for Integration

The Exchange Negotiation Service, aka the **Contract Proxy** system, 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.

16.2 Technical Details

The Contract Proxy system consists of four primary components, illustrated in following figure, which are 1) the Negotiation Service (NS), 2) the User Registry (UR), 3) the Exchange Ledger (EL) and 4) the Definition Bank (DB).

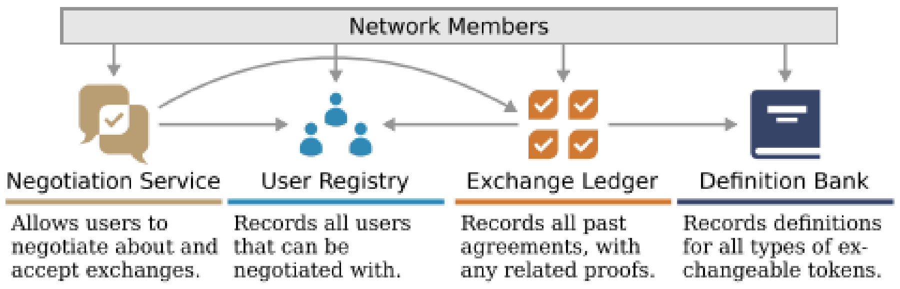

Network Members
Negotiation Service
Allows users to negotiate about and accept exchanges.
User Registry
Records all users that can be negotiated with.
Exchange Ledger
Records all past agreements, with any related proofs.
Definition Bank
Records definitions for all types of ex-changeable tokens.

Figure 16.1 The components of the Contract Proxy system.

The recording of past agreements can be implemented in a centralized system, with a distributed ledger technology (DLT), aka BlockChain, or any other suitable way. LTU has chosen to implement the Exchange Ledger with signature chains that are kept by negotiating parties only to preserve privacy. That is, each signature chain can be kept private in between parties that uses the Contract Proxy to interact. As shown in the following picture, the Negotiation service is kept simple to provide support for negotiation only. More advanced logic is thereby to be implemented by each party, outside the Contract Proxy system.

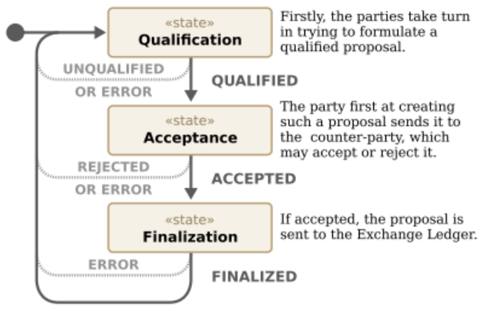

«state» Qualification
Firstly, the parties take turn in trying to formulate a qualified proposal.
UNQUALIFIED OR ERROR
QUALIFIED
«state» Acceptance
The party first at creating such a proposal sends it to the counter-party, which may accept or reject it.
REJECTED OR ERROR
ACCEPTED
«state» Finalization
If accepted, the proposal is sent to the Exchange Ledger.
ERROR
FINALIZED

Figure 16.2 The Contract Proxy Negotiation service.

16.3 Integration in Arrowhead Framework

Plan to include in the Arrowhead Framework as part of the continued work.

16.4 Status of Work

The service is being prepared for final-review demonstration in the Productive 4.0 project. This demonstration aims at showing the use of contractual interactions supported by the Contract Proxy system for a supply-chain scenario with financial forecasting based on the trail of data kept by the system as well as the ordering of production, including negotiation of price and volumes. In addition, the demonstration may include contractual interactions for handing out certificates at onboarding of devices into an Arrowhead Local cloud. As illustrated in the following figure, the demonstrations include integration with the Event Handler system, Gateway systems and the Certificate Authority system.

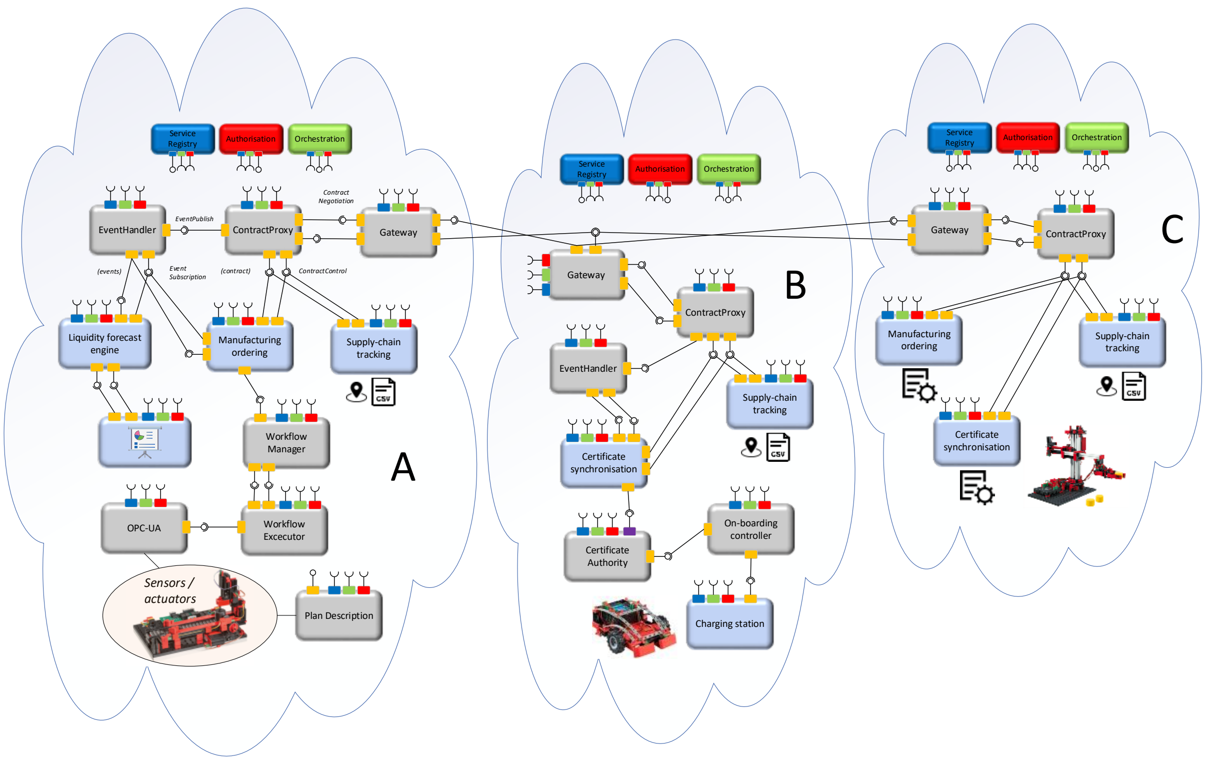


Figure 16.3 Demonstration of three different usages of the Contract Proxy system.

For the Contract Proxy system, the Arrowhead Framework documentation is being prepared to follow the updates of the implementation. The work on this system will be continued as part of Arrowhead Tools.

17 Secure data sharing (new concept)

17.1 Rationale for Integration

Automated and secure sharing of data from time series databases is becoming increasingly important for the industry, e.g. to gain from analyzing expertize outside the own organization, to support AI and machine learning in between organization, or to share information to facilitate remanufacturing, refurbishing and reuse of products and materials in a circular value-chain. The needs for automated and secure data sharing applies to essentially any data, including time series data. In this work, LTU has focused on the use of attribute based access control (ABAC) systems and their ability to onboard new data streams and control access to data. The systems evaluated are the Next-Generation Access Control (NGAC) developed by Institute of Standards and Technology (NIST) and the Extensible Access Control Markup Language (XACML) by the Organization for the Advancement of Structured Information Standards (OASIS). The results of this work are documented in a paper published at IEEE EFTA 2020 [37].

17.2 Technical Details

Findings presented in the above-mentioned paper include that NGAC supports automation of data stream onboarding and access control at the level of individual data streams but need to be extended to also support more fine-grained access control with line-based access restrictions to the data. Such restrictions are needed for example to easily automate the access control for giving access only to data before a certain date and time and block access to newer data. As illustrated in the following figure, the experimental system set up implements pre-filtering of queries to perform the desired access control.

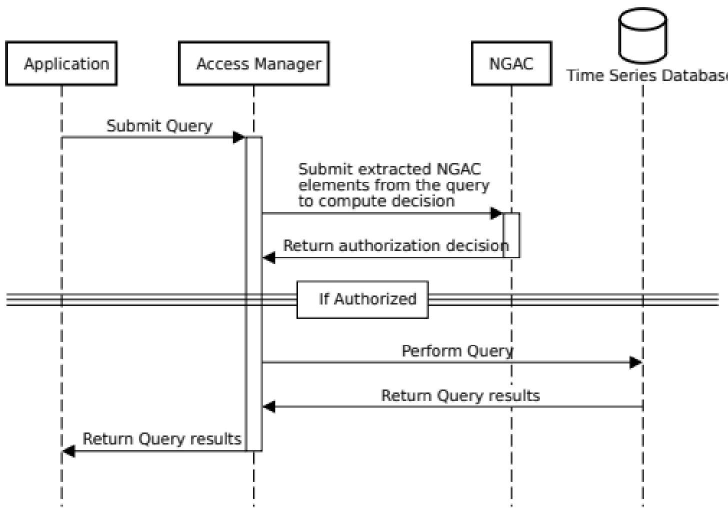


Figure 17.1 Pre-filtering of queries to time-series data.

17.3 Integration in Arrowhead Framework

The aim is to later integrate the demonstration with the Arrowhead Framework as a tools for secure data sharing.

17.4 Status of Work

Ongoing work include setting up a proof-of-concept demonstration for automated access control to time series data based on NGAC.

18 Plant Description Engine (from Productive 4.0)

18.1 Rationale for Integration

The Plant Description Engine (PDE) is a supporting core system that has been developed during the period, starting with a conceptual model and a driving use case where a plan (i.e., a plant description) is established to deploy and interconnect predefined AH systems in a multi-vendor scenario. The PDE has a purpose of choreographing the consumers and producers in the plant (System of Systems / Local cloud) by interaction with the Orchestrator to allow systems to connect (using late binding) according to the plan. The PDE performs monitoring to oversee that the plan is followed and can produce alarms when something falls out of plan.

18.2 Technical Details

The PDE can receive a number of plans. A plan (i.e., a plant description) is an abstract view on which systems the plant should contain and how they should be connected as consumers and producers. The plan is used to populate the Orchestrator with store rules for each of the consumers. The abstract view does not contain any instance specific information, instead meta-data about each system is used to identify the service producers.

The plant description engine (PDE) can be configured with several variants of the plant description of which at most one can be active. The active plant description is used to populate the orchestrator and if no plant description is active the orchestrator does not contain any store rules populated by the PDE. This can be used to establish alternativ plants (plan A, plan B, etc).

The PDE gathers information about the presence of all specified systems in the active plant description. If a system is not present it raises an alarm. If it detects that an unknown system has registered a service in the service registry it also raises an alarm. For a consumer system to be monitored the system must produce the Monitorable service and hence also register in the service registry.

18.3 Integration in Arrowhead Framework

The PDE has been integrated with Arrowhead. Please see the Plant Description Engine - System of systems Description (SosD) and Plant Description Engine HTTP(S)/JSON - System Description (SysD) for further details.

The PDE produces two different services:

- the Plant Description Management service - Plant Description Management JSON
- the Plant Description Monitor service - Plant Description Monitor JSON

The PDE consumes the following services:

- the Service Discovery service produced by the Service Registry core system
- the Orchestration Store Management service produced by the Orchestrator core system
- the Orchestration service produced by the Orchestrator core system
- the Inventory service produced by an Inventory system - Inventory JSON
- the Monitorable service produced by the systems in the plant - Monitorable JSON

18.4 Status of Work

A proof-of-concept for demonstration is ready. This demo is under the assumption that some functions in the core systems are in place, which will be in future versions of those systems.

19 Authorization by Power of Attorney (new concept)

19.1 Rationale for Integration

Distributed Internet of Things and cyber-physical systems have the potential to act on behalf of their owners. To accomplish this in a controlled way, there is a need for a model where such devices can be given exact credentials and authorities for the specific actions they should be taking. Traditional models where each device has an account of its own at various service points to regulate its credentials have scalability problems in management of all the accounts and credentials. Instead, we propose a model based on principals and agents, where principals can independently define from case to case which credentials and powers to delegate. The results of this work are documented in a paper published at IEEE EFTA 2020 [41].

19.2 Technical Details

In this work LTU has defined the conceptual model and an early proof of concept, where we introduce the Power of Attorney (PoA), which is a self-contained and signed digital document that for a limited time and in a defined context, authorizes a particular agent (whether a person or device) to sign on behalf of a principal. Such a PoA includes digital signatures based on public and private keys to identify the parties, while the key content of a PoA is to specify exactly what the principal authorizes the agent to perform on its behalf. An essential property is that the PoA is self-contained, so that it can be brought forward by an agent to a service in order to use the accounts and some well-defined sub-parts of the authorities of its principal, for example to retrieve or produce data on behalf of its principal.

Although such self-contained PoAs can be stored anywhere, we also design a conceptual model for a signatory registry that can store PoAs and keep track of organizational hierarchies in terms of people and devices according to the defined model.

19.3 Integration in Arrowhead Framework

The aim is to later integrate the demonstration with the Arrowhead Framework as a tool for delegating authority.

19.4 Status of Work

Ongoing work include setting up a proof-of-concept for demonstration.

20 Software Deployment and virtualization (new concept)

20.1 Rationale for Integration

Distributed Internet of Things and cyber-physical systems are networked and to a large extent software defined. This means that devices are becoming more general purpose and that their exact functionality is defined by software running onboard and in the cloud. In this context, updates and upgrades of functionalities is mostly done by deploying software to devices.
To accomplish this in a controlled way, there is a need to automate the deployment of software and to support verification and validation of the functionalities.

The results of this work are documented in a paper that is under submission:
”A Study on Industrial IoT for Mining Industry: Synthesized Architecture and Open Research Directions”

20.2 Technical Details

In this work the starting point was to explore the needs in industrial IoT in general, and specifically analyzing the current state and challenges of the mining industry. We identified the vertical fragmentation and lack of automation as key issues. We have surveyed the IIoT standards and architectures suitable for the mining industry. From these, we synthesize a high-level IIoT architecture. We have identified the open research challenges clearly and depict that the edge and fog computing is the most challenging layer for the mining industry and the same is truefor various industrial sectors. The above mentioned challenges for the edge can be addressed with the advancements in the virtualization techniques for edge computing. The virtualization-based solution in the edge computing for deployment, orchestration, updates, and upgrades are highly needed. Not only the mining industry but almost all the industries are quite complex at the edge with the advancement of IoT based microservices, and systems. There are a great number of solutions available for cloud computing orchestration and deployment of services, but a standard or concrete solution targeting the same problem for the edge/fogdoes not exist.

20.3 Integration in Arrowhead Framework

The aim is to later make a Proof-of-concept for edge virtualization and software deployment for applications within the Arrowhead Framework.

20.4 Status of Work

Ongoing work include initial suproof-of-concept for demonstration.

21 Workflow Management/Executor (from Productive 4.0)

21.1 Rationale for Integration

Important to address the execution of a product assembly workstation operations based on the product recipe. Thus enabling production flexibility and lot-size one capabilities in a production line.

21.2 Technical Details

The Workflow Management provides integration with ERP systems to collect production order and translate them to state machines for manufacturing. The state machines are then transferred to different Workflow Execution systems that integrate with different tools to be used for the manufacturing, effectively hiding manufacturing and tool details. As illustrated in the following figure, this concept facilitates scenarios where more advanced product carry their own recipes for how they shall be produced in a manufacturing or assembly line.

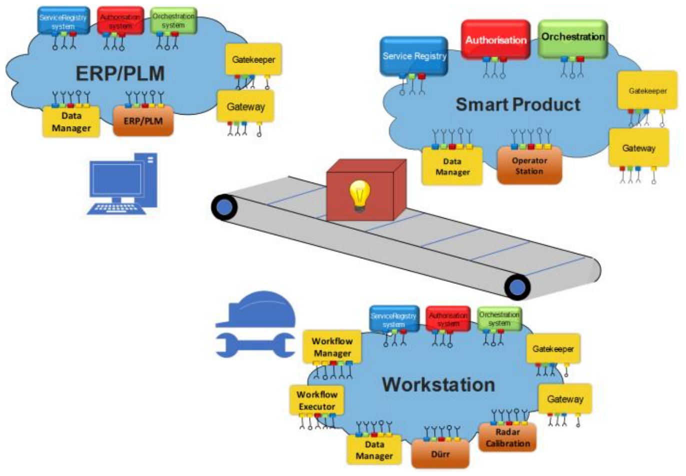


Figure 21.1 The Smart Product, the Workstations and their interactions in the Workflow Management/Executor concept

As shown in the figure, with the Workflow Manager/Executor systems, a plant can implement decentralized and distributed workflows.

21.3 Integration in Arrowhead Framework
Currently at prototype status

21.4 Status of Work
Working code and integration to PlantDescription and Exchange Network in the works. The prototype can receive and translate a product production recipe into a work order for a workstation in a production line. This work order can then be executed.
The current GAP is how to address product recipes based on a number of various standards, like BIM v5, ISO 10303, ISO 15926 etc.

22 Workflow Choreographer / Workflow Executor
In order to manage and execute workflow recipes in production and logistics, The Workflow Choreographer supporting core system was defined within Arrowhead. There are a lot of requirements and restraints the Workflow Choreographer faces upon the management and execution of workflows, such as constraints of workstations that are used, the sequencing (sometimes parallelisms) of various tasks and services, and further aspects of production plants. It was first introduced as an engine that controls automated production and in doing so was planned to greatly reduce the difficulty of management challenges in a production facility.

The features and services that the Workflow Choreographer provides can be seen on the below figure.

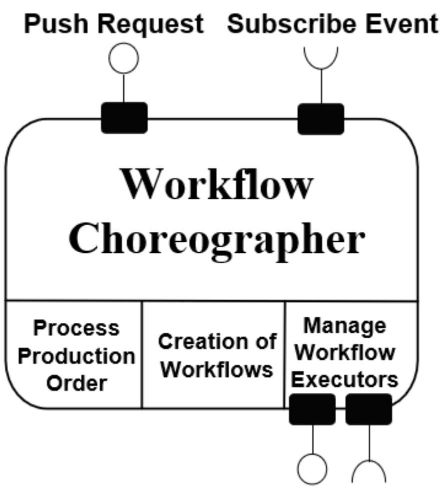


Figure 22.1 The functions and services of the Workflow Choreographer

First it processes the Production Order (PO) acquired from various Enterprise Resource Planning (ERP) systems and creates the Production Recipe (PR) based on this data. Furthermore, it manages the different Workflow Executors (WE) to accomplish the tasks at hand either by instantiating them or sending messages to already instantiated WEs. After getting information about the production steps in a recipe the WEs execute the related activities by coordinating the workstations to perform the modifications described in the PR. From the Arrowhead Framework's point of view the Choreographer pushes requests to the Orchestrator and subscribes to events reported by other application systems.

The Workflow Choreographer needs to store information about every workflow running in the system to ensure their proper execution by monitoring, detecting and handling possible errors occurring during production. To achieve this, two core systems support the workflow Choreographer, namely the Orchestrator and the Event Handler (EH). The former enables the Choreographer to pair application systems (service consumers and providers) together while the latter delivers messages across the system if something goes wrong or a workstation finished task execution successfully.

The below figure shows a simple workflow for the Choreograper [42]. The Choreographer consumes the service of the Orchestrator for pairing the two application systems for starting task execution. Then the WE takes control and manages the application systems to achieve the behavior described by the production recipe. This is followed by a callback to the Choreographer with the result (either if the task is done or some unexpected error happened that needs handling).

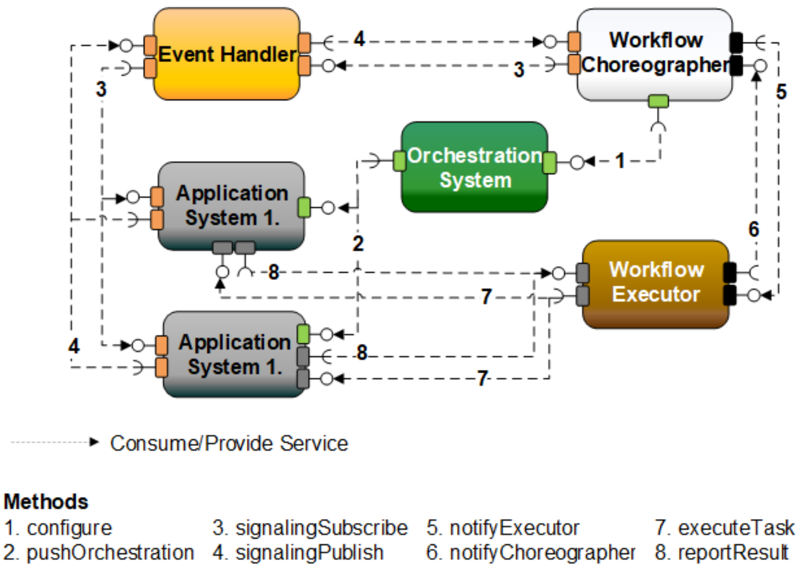


Figure 22.2 Services consumption interactions in a the simple scenario when the Workflow Choreography and Workflow Executor working together

Workflow Choreographer is a workflow governing unit in the Arrowhead Framework. It can create workflows from the Production Orders and allocate tasks to the *Workflow Executors* for execution while logging relevant data about every phase of the production.

The initial concept of the Workflow Choreographer in Arrowhead has aimed to execute only predefined tasks in static environments. In the latest period we went further and added new functions and features to the system making it capable of working in dynamic scenarios where occasionally some devices go offline even during execution and another device must step in to accomplish the task [43].

The main new element introduced in this part of our work is the creation of Workflow Executors (WEs) and their integration with the Workflow Choreographer. WEs make it possible to establish connection between the AH core systems (Orchestrator, Choreographer) and the

Application Systems (application service providers, consumers). In the previous Choreography versions the providers communicated with the Choreographer directly, which introduced significant dependency from the AH's point of view. Services provided by the Arrowhead Framework core systems should not introduce future implementation difficulties for the Application Systems therefore the independence between the core systems and the various application service providers was cut to almost zero by using WEs.

22.1 Rationale for Integration

Workflow Choreography and Execution is in the main stream of Arrowhead Tools, because it supports production workflow planning, recipe execution in a dynamic manner.

22.2 Technical Details

To highlight the new contribution for Workflow Choreography and Execution, the below figure shows the high-level architecture of a Local Cloud with the Arrowhead Core Systems, the Application Systems, including the Workflow Executors and the various services they provide each other.

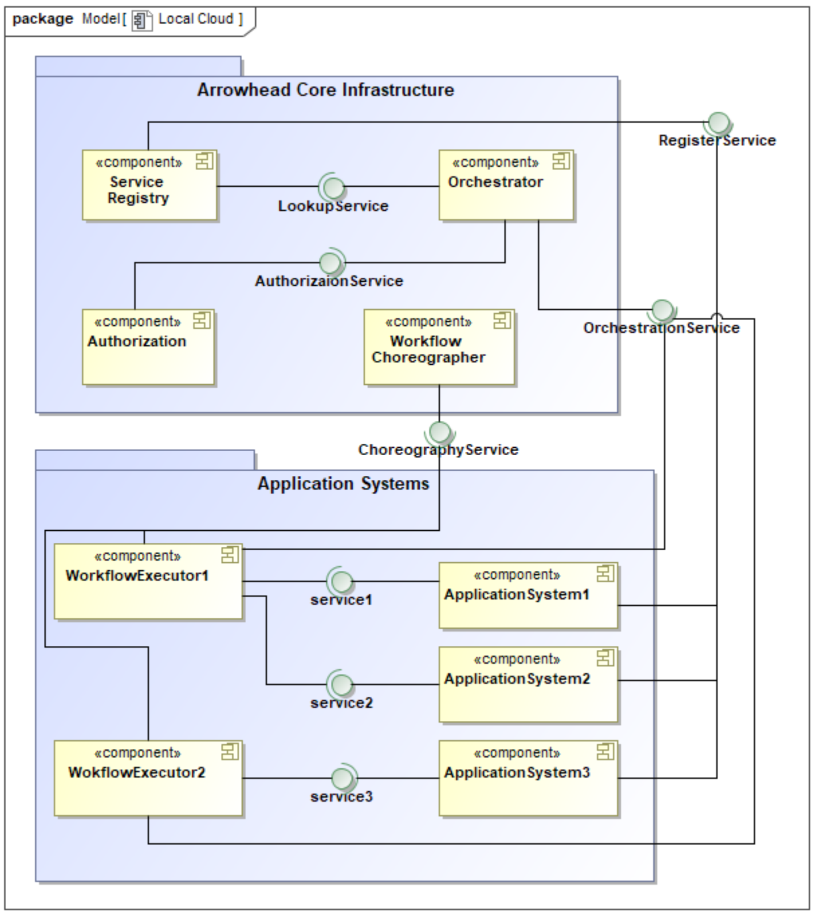

package Model[Local Cloud]
Arrowhead Core Infrastructure
RegisterService
«component» Service Registry
LookupService
«component» Orchestrator
AuthorizaionService
«component» Authorization
«component» Workflow Choreographer
OrchestrationService
ChoreographyService
Application Systems
«component» WorkflowExecutor1
service1
«component» ApplicationSystem1
«component» ApplicationSystem2
service2
«component» WokflowExecutor2
service3
«component» ApplicationSystem3

Figure 22.3 Local Cloud model with the Workflow Choreographer and Executors

To accomplish the execution of each task in the recipe, the Choreographer sends a request to the suitable Executor. In this case the WEs can be comprehended as special consumers by pushing the request they get from the Choreographer to the corresponding providers, in other words forcing these providers to execute tasks the Choreographer previously allocated to them according to the production recipe by service consumption. The WE can monitor the status of the tasks under execution in various ways e.g., by periodically checking the blinking of a led sensor on the device, polling the providers regularly or waiting for callback messages from the providers executing the task.

A more complex behavior – fitting the shown local cloud – is depicted by the below figure, which shows the sequence of messages and method calls during workflow execution.

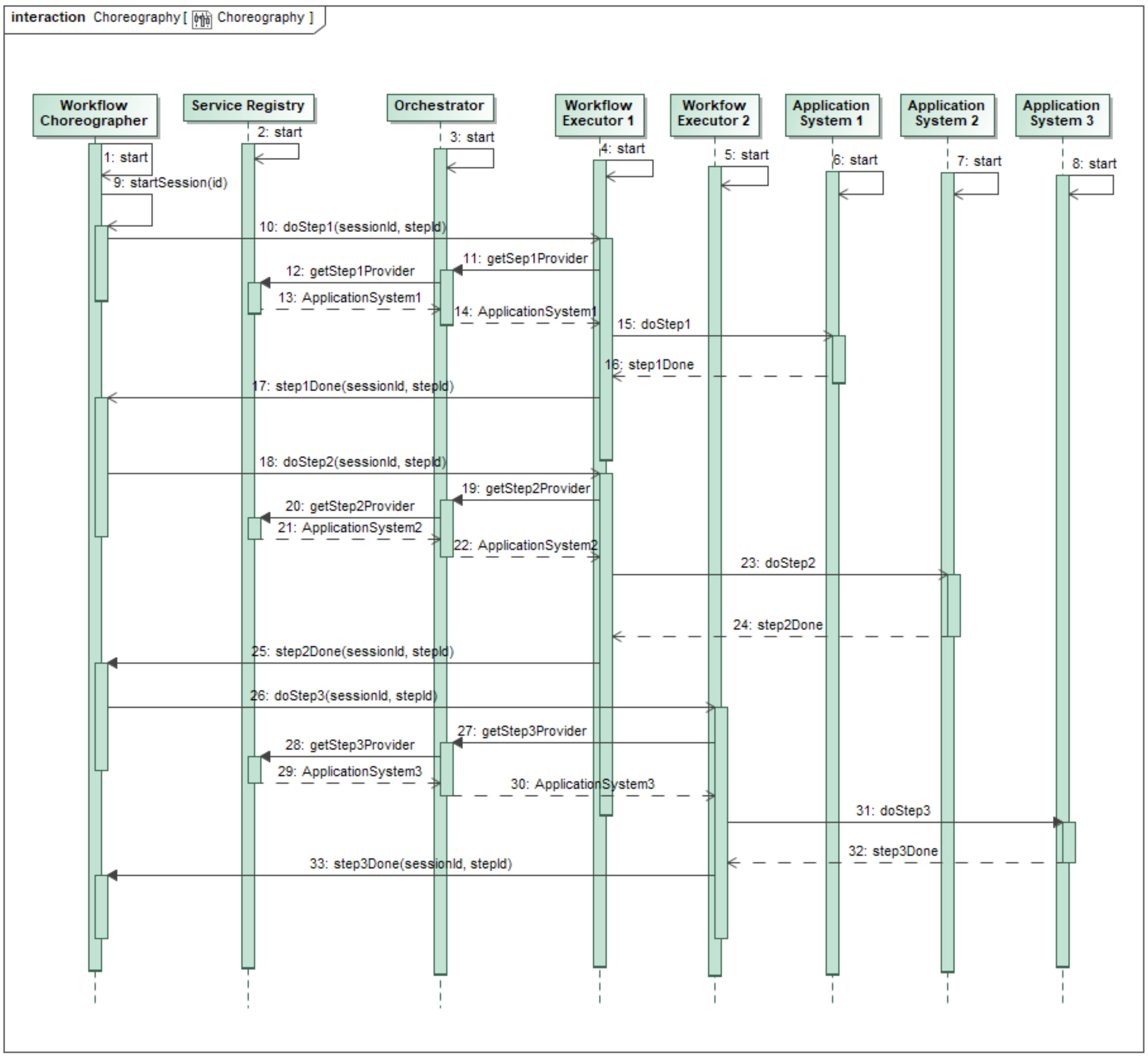


Figure 22.4 An example message sequence for the Local Cloud with the Workflow Choreographer and the Executors

When the Choreographer is tasked to manufacture a product, first it checks if the Local Cloud contains all the necessary providers and Workflow Executors according to the recipe. If it finds that the product could not be manufactured under the current circumstances then a system operator is warned to fill in the gaps by either adding the missing provider systems to the environment or by extending the functions of the current WEs. In some cases even adding and implementing new WEs may be needed.

22.3 Integration in Arrowhead Framework

The Workflow Choreographer has become a fully released Arrowhead supporting core system with version 4.2. The Workflow Executor works with the Choreographer in harmony as a prototype, but it has not been fully reviewed and released yet. Nevertheless, the Workflow Executor acts as an Application system.

22.4 Status of Work

The Choreographer is released together with other v4.2 Arrowhead core systems; the Workflow Executor is being reviewed and fully integrated with the next release version.

23 Energy IoT Monitoring Platform

The Arrowhead Framework can be used in a Smart Energy use case scenario to implement an IoT monitoring platform for smart meters. In order to operate a large fleet of sensors across the region, managers and maintainers need advanced tools to control and monitor the status of edge devices. The main purpose of the proposed system is to support those figures in the management and maintenance of smart meters by providing an integrated platform to collect and visualize information about the status of peripheral devices. This kind of information can enable the creation of more advanced analytics to track and monitor the coverage of smart meters, together with their activities. Furthermore, information about locality of individual devices can be meaningful to produce data analytics aggregated by location, thus allowing to monitor the energy consumptions of specific geographical areas.

23.1 Rationale for Integration

The Arrowhead Framework will provide basic functionalities for service discovery, orchestration and authorization. The information about the status of smart meters together with their power measurements constitute the services provided by our application and can be consumed by interacting with the Arrowhead Framework Core Systems (service registry, orchestrator, authorization). The following picture shows the architecture of the proposed monitoring platform.

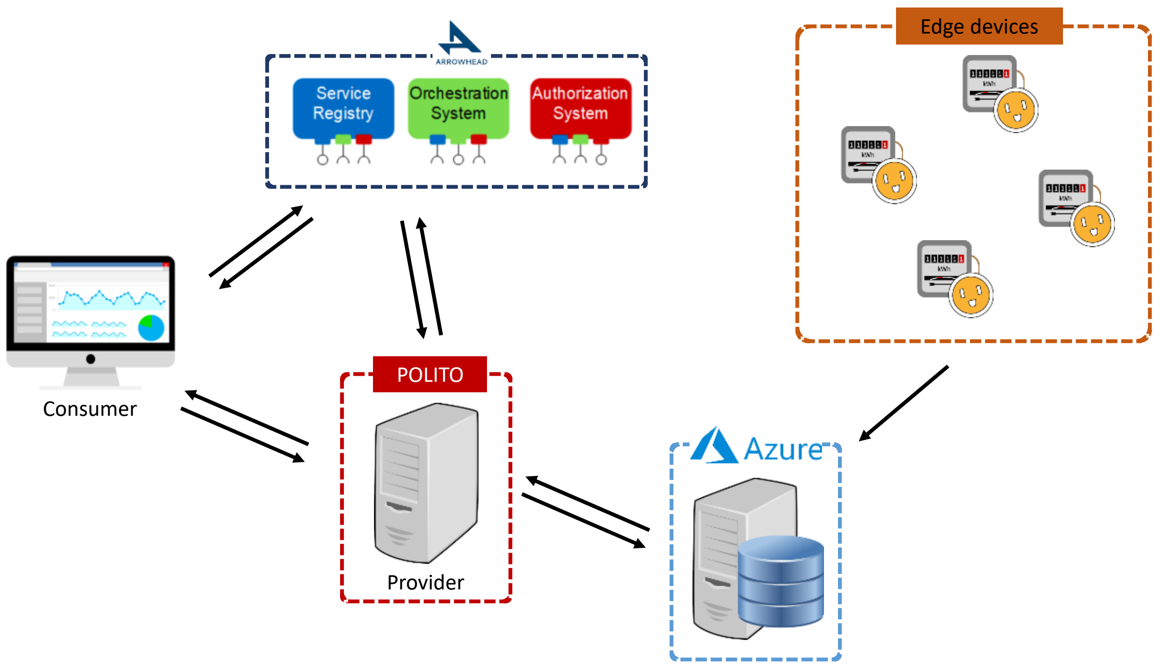


Figure 23.1 The architecture of the proposed Arrowhead-compliant monitoring platform

23.2 Technical Details

The monitoring platform consumes status information and energy data from the edge devices of an existing legacy infrastructure and supplies them to potential third-party consumers by using the Arrowhead Framework. The legacy infrastructure is provided by a third-party stakeholder operating in the energy sector, which already owns a fleet of sensors deployed across several households. The sensors operate in the low frequency range with a sampling frequency of 1 Hz and periodically send their status information to the cloud. The data received from smart meters are preprocessed and permanently stored in a Microsoft Azure database. The database represents the interface between the legacy infrastructure and the monitoring platform, exposing all available status information and energy data to the Arrowhead Framework through the database API.

23.3 Integration in Arrowhead Framework

The monitoring platform is integrated into the Arrowhead Framework by extending the Java Spring client-skeletons provided by the Arrowhead Consortia. The monitoring platform provides an interface between the Arrowhead Framework and an already existing legacy infrastructure. The platform comes up with a configurable web dashboard which extends the functionalities of the consumer application, thus facilitating data access from managers and maintainers.

The Arrowhead Framework will be extended with additional services for monitoring the status of edge devices during operation. Information such as battery status, device’s activities and location can be retrieved and combined together to produce useful analytics for efficiently maintaining the fleet of sensors. Moreover, energy data can be aggregated by location in order to perform aggregate analysis for specific regions. All information will be provided through the implementation of parametrized APIs and web dashboards.

23.4 Status of Work
The Politecnico di Torino (Polito) has defined the requirements and the functional design of the monitoring platform, in accordance with the third-party stakeholder owning the legacy infrastructure. In the next months the application logic and the web dashboard will be implemented to support the required functionalities.

24 Arrowhead Framework on STM32
The STM32 family of 32-bit microcontrollers based on the Arm® Cortex®-M processor is designed to offer new degrees of freedom to MCU users. It offers products combining very high performance, real-time capabilities, digital signal processing, low-power / low-voltage operation, and connectivity, while maintaining full integration and ease of development [40].

24.1 Rationale for Integration
STM32 boards constitute one of the main products family of STMicroelectronics and they have been used in smart Energy scenario and other usecases. It will also used in WP6 activity.

24.2 Technical details
The activity aims to develop an entire deployment unit based on STM32 platforms. The unit will be a local cloud, which is an autonomous set of devices that have access to an Arrowhead Framework (represented in Figure).

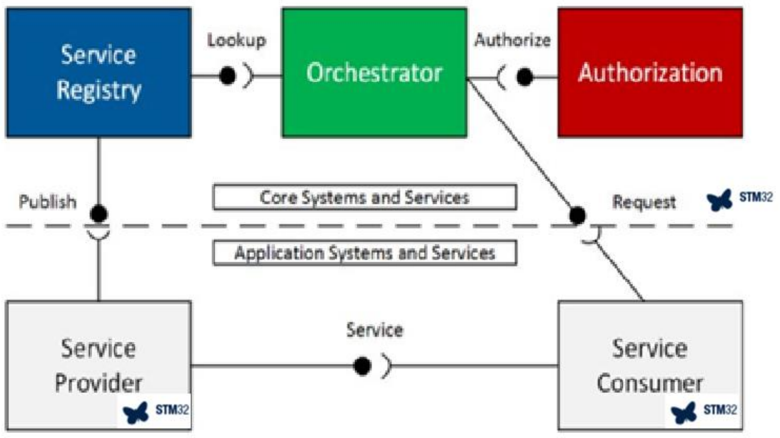


Figure 24.1 Main elements of an Arrowhead Local Cloud with STM-32 Application Systems

The implementation will be a customization of the Arrowhead framework tailored for the embedded platform, in particular a single jar implementation optimized in terms of memory footprint, and execution time.

24.3 Integration in Arrowhead Framework
Currently at prototype status

24.4 Status of Work
Selection of the board for the framework on going. Development of consumer and producer code in the works.

25 Keycloak integration

Keycloak [28] is an open source Identity and Access Management solution aimed at modern applications and services. It makes it easy to secure applications and services with little to no code.

This tool supports:

- Single-Sign on

  Users authenticate with Keycloak rather than individual applications. This means that the applications don't have to deal with login forms, authenticating users, and storing users. Once logged-in to Keycloak, users don't have to login again to access a different application. This also applied to logout. Keycloak provides single-sign out, which means users only have to logout once to be logged-out of all applications that use Keycloak.

- LDAP and Active Directory, existing user directories can be used for corporate use

- Standard Protocols

  OpenID Connect, OAuth 2.0, SAML 2.0, which are widely used in the industry

- Centralized Management for admins and users via Account Management Console

  Through the account management console users can manage their own accounts. They can update the profile, change passwords, and setup two-factor authentication.
  Users can also manage sessions as well as view history for the account.

- Custom password policies

- Extensions through custom code

25.1Rationale for Integration

The Management Tool which is interfacing with the Arrowhead Framework is capable of the management of Local Arrowhead Clouds. But the Tool itself does not offer any authentication or authorization methods. This way anyone is capable of accessing high level Arrowhead functions. This poses security risks which is unacceptable in the industrial world. Integrating Keycloak into the Management Tool for this reason, offers an elegant solution for this problem.

25.2Technical Details

The implementation has two parts:

1. Installing and configuring Keycloak to a host machine. They provided guide by Keycloak was followed during the setup. Furthermore, all setup regarding the user authentication / authorization and role management has to be made on the Keycloak side

2. Interfacing with Keycloak in the Management Tool; the below figure shows the complete process.

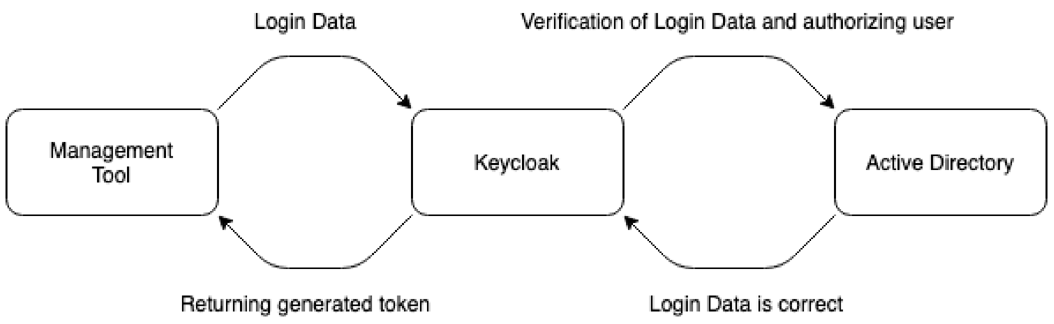

Login Data
Verification of Login Data and authorizing user
Management Tool
Keycloak
Active Directory
Returning generated token
Login Data is correct

Figure 25.1 Authorization flow with Keycloak

The Authorization flow starts with the user opening the Management Tool. The Management Tool checks for the existence of the authentication token. If no authentication token is found, the user needs to authenticate itself. Login data must be provided for Keycloak by the user. Then Keycloak contacts the company's LDAP or Active Directory based database to verify the login information. If the login data is correct then Keycloak generates a token for the user. This token contains all necessary information for the Management Tool.

These tokens are JSON Web Tokens (https://jwt.io/). If we further inspect the token, we can see what information does it contain.

- Issue time
- Expiry time
- Issuer
- Authorized Party
- Allowed-Origin
- Roles
- Email
- Family Name, Given Name

With this information the Management Tool can validate the token, moreover it is capable of dynamically disable functionality based on the role of the user. A practical example is, that a technician has only read only right (displaying data on graphs and in table views) while the System Operator has full access to every part of the framework.

25.3Integration in Arrowhead Framework

Since Arrowhead Framework does not have User Management, it only uses X.509 Certificates for client authentication
The Arrowhead Management Tool is a widely used tool by many parties in the consortium. This enhancement to the Tool allows them to limit access and secure their instances.

25.4Status of Work

Available Features:

- User Authentication in Management Tool with the help of Keycloak
- Role management
- LDAP, Active Directory integration

Present Gaps:

Fine tuned role management, further limiting access for different core systems in the Management Tool.

26 Conclusions

The core concepts of the Arrowhead Framework keeps expanding and maturing. The long-term governance model is successful so far: more and more Eclipse IoT projects consider to collaborate with parts oft he Arrowhead Tools project, actively. The proceedings of its conceptual growth, its design and implementation are summarized within this document. There are three 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.

Third, a very visible part of this document: Eclipse IoT projects actively finding collaboration and integration possibilities with Arrowhead, and this is possible now on the re-usable open-source code level as well, due to the fact that Arrowhead recently became an approved, continuously reviewed, governed project under the Eclipse license.

27 References

[1] Eclipse IoT Working Group, "The Three Software Stacks Required for IoT Architectures," 2016. [Online]. Available: https://iot.eclipse.org/community/resources/white-papers/pdf/Eclipse IoT White Paper – The Three Software Stacks Required for IoT Architectures.pdf.

[2] E. Foundation, "Eclipse IoT," 2019. https://iot.eclipse.org/ (accessed Apr. 25, 2020).

[3] Eclipse IoT Working Group, "Open Source Software for Industry 4.0," 2017. [Online]. Available: https://iot.eclipse.org/resources/white-papers/Eclipse IoT White Paper – Open Source Software for Industry 4.0.pdf.

[4] "Eclipse Vert.x." https://vertx.io/ (accessed Apr. 25, 2020).

[5] "Spring." https://spring.io/ (accessed Apr. 25, 2020)

[6] "Eclipse Vorto." https://www.eclipse.org/vorto/ (accessed Apr. 25, 2020).

[7] J. Delsing, *IoT automation: Arrowhead framework*. CRC Press, 2017.

[8] C. Hegedus, P. Varga, and A. Franko, "Secure and trusted inter-cloud communications in the arrowhead framework," *Proc. – 2018 IEEE Ind. Cyber-Physical Syst. ICPS 2018*, pp. 755–760, 2018, doi: 10.1109/ICPHYS.2018.8390802.

[9] O. Carlsson, D. Vera, J. Delsing, B. Ahmad, and R. Harrison, "Plant descriptions for engineering tool interoperability," *IEEE Int. Conf. Ind. Informatics*, vol. 0, pp. 730–735, 2016, doi: 10.1109/INDIN.2016.7819255.

[10] Open Mobile Alliance, "Lightweight M2M (LWM2M) – OMA SpecWorks." https://www.omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/ (accessed Jan. 19, 2020).

[11] "Eclipse Leshan." https://www.eclipse.org/leshan/ (accessed Apr. 25, 2020).

[12] O. Carlsson *et al.*, "Configuration service in cloud based automation systems," *IECON Proc. (Industrial Electron. Conf.*, pp. 5238–5245, 2016, doi: 10.1109/IECON.2016.7793489.

[13] "Eclipse Wakaama." https://www.eclipse.org/wakaama/ (accessed Apr. 25, 2020).

[14] "Eclipse Keti." https://projects.eclipse.org/proposals/eclipse-keti (accessed Apr. 25, 2020).

[15] "Eclipse Mosquitto." https://mosquitto.org/ (accessed Apr. 25, 2020).

[16] "Eclipse Paho." https://www.eclipse.org/paho/ (accessed Apr. 25, 2020).

[17] A. Zabasta, K. Kondratjevs, J. Peksa, and N. Kunicina, "MQTT enabled service broker for implementation arrowhead core systems for automation of control of utility' systems," *Proc. 5th IEEE Work. Adv. Information, Electron. Electr. Eng. AIEEE 2017*, vol. 2018-Janua, pp. 1–6, 2017, doi: 10.1109/AIEEE.2017.8270543.

[18] “Eclipse Ditto.” https://www.eclipse.org/ditto/ (accessed Apr. 25, 2020).

[19] H. Derhamy, J. Eliasson, and J. Delsing, “IoT Interoperability – On-Demand and Low Latency Transparent Multiprotocol Translator,” *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1754–1763, 2017, doi: 10.1109/JIOT.2017.2697718.

[20] “Eclipse hawkBit.” https://www.eclipse.org/hawkbit/ (accessed Apr. 25, 2020).

[21] “Eclipse Hono.” https://www.eclipse.org/hono/ (accessed Apr. 25, 2020).

[22] M. Jones, J. Bradley, and N. Sakimura, “RFC 7519: JSON Web Token (JWT),” 2015.

[23] M. Jones, E. Rescorla, and J. Hildebrand, “RFC 7516: JSON Web Encryption (JWE),” 2015.

[24] M. Jones, J. Bradley, and N. Sakimura, “RFC 7515: JSON Web Signature (JWS),” 2015.

[25] “OpenID Connect.” https://openid.net/connect/ (accessed Apr. 25, 2020).

[26] D. Hardt, “RFC 6749: The Oauth 2.0 Authorization Framework,” 2012.

[27] B. Campbell, J. Bradley, N. Sakimura, and T. Lodderstedt, “RFC 8705: Oauth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens,” [Online]. Available: https://tools.ietf.org/html/rfc8705.

[28] “Keycloak.” https://www.keycloak.org/ (accessed Apr. 25, 2020).

[29] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, “The Reactive Manifesto,” *Reactivemanifesto.Org*, 2014.

[30] G. Jansen and P. Gollmar, *Reactive Systems Explained*. O’Reilly Media, Inc., 2020.

[31] “TechEmpower Framework Benchmarks.” https://www.techempower.com/benchmarks/ (accessed Apr. 25, 2020).

[32] C. Hewitt, P. Bishop, and R. Steiger, “A Universal Modular ACTOR Formalism for Artificial Intelligence,” 1973, doi: 10.1145/359545.359563.

[33] N. Naik, “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP,” 2017, doi: 10.1109/SysEng.2017.8088251.

[34] A. Melnikov and K. Zeilenga, “RFC 4422: Simple Authentication and Security Layer (SASL),” 2006, [Online]. Available: https://tools.ietf.org/html/rfc4422.

[35] K. Hartke, “RFC 7641: Observing Resources in the Constrained Application Protocol (CoAP),” 2015.

[36] L. Römer, S.E. Jeroschewski, and J. Kristan, „Leveraging Eclipse IoT in the Arrowhead Framework,” *NOMS 2020 – 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE Press, 2020.

[37] Chiquito, A., Bodin, U., & Schelén, O. (2020). Access Control Model for Time Series Databases using NGAC. Presented at the 25th International Conference on Emerging Technologies and Factory Automation, Vienna, September 8-11, 2020.

[38] J Nilsson and F Sandin, *Semantic Interoperability in Industry 4.0: Survey of Recent Developments and Outlook*, 2018 IEEE 16th International Conference on Industrial Informatics (INDIN), pp. 127-132.

[39] J Nilsson, F Sandin, J Delsing, *Interoperability and machine-to-machine translation model with mappings to machine learning tasks*, 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), pp. 284-289.
J Nilsson, J Delsing and F Sandin, *Autoencoder Alignment Approach to Run-Time Interoperability for System of Systems Engineering*, 2020 IEEE 24th International Conference on Intelligent Engineering Systems (INES), pp. 139-144

[40] https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[41] Sudarsan, S. V., Schelén, O., and Bodin, U, *A Model for Signatories in Cyber-Physical Systems*. Presented at the 25th International Conference on Emerging Technologies and Factory Automation, Vienna, September pp. 8-11, 2020.

[42] P. Varga, D. Kozma, and C. Hegedűs, *Data-Driven Workflow Execution inService Oriented IoT Architectures*. In2018 IEEE 23rd International Conference onEmerging Technologies and Factory Automation (ETFA), volume 1, pages 203–210.IEEE, 2018

[43] D. Kozma, P. Varga, and K. Szabó. *Achieving Flexible Digital Productionwith the Arrowhead Workflow Choreographer*. The 46th Annual Conference of theIEEE Industrial Electronics Society (IECON), October 2020

28 Revision history

28.1 Contributing and reviewing partners

Contributions	Reviews	Participants	Representing partner
Chapter 2, Arrowhead Framework Mandatory Core – v4.2	LTU	Pal Varga, BME; Szvetlin Tanyi, AITIA	BME, AITIA
Chapter 3, Eclipse IoT Integration	BME	Johannes Kristan, Bosch.IO GmbH; Lukas Römer, Bosch.IO GmbH	Bosch.IO GmbH
Chapter 4 Eclipse Kapua and Kura	Bosch, BME	Paolo Azzoni, Eurotech	Eurotech
Chapter 5 Onboarding Procedure, Chapter 6 Monitoring and Standard Compliance Verification	Bosch, BME	Silia Maksuti, Forschung Burgenland	Forschung Burgenland
Chapter 7 Vital IoT	Bosch, BME	Gerry Nigro, Concept Reply	Concept Reply
Chapter 8 IKERLAN Tool Adapter	Bosch, BME	Fernando Eizaguirre, IKERLAN	IKERLAN
Chapter 9 Extended Historian Service	Bosch, BME	Mario Thron, ifak	IKT & Automation ifak e.V. Magdeburg
Chapter 10 WAE (Web-of-Things Arrowhead Enabler	Bosch, BME	Federico Montori, UNIBO	University of Bologna, Italy
Chapter 11 Python Client Library, Chapter 12 Semantics translator , Chapter 13 Code generation, Chapter 14 Integration of OPC-UA server with late binding, Chapter 15 Translator system, Chapter 16 Exchange Negotiation Service (from Productive 4.0), Chapter Secure data sharing (new concept), Chapter 17 Secure Data Sharing Chapter 18 Plant Description Engine Chapter 19 Authorization by Power of Attorney 20 Software Development and virtualization Chapter 21 Workflow Management /Executor	Bosch, BME	Ulf Bodin, Lulea Technical University	Lulea Technical University

Chapter 22 Workflow Choreographer / Workflow Executor	LTU	Pal Varga, BME Kristóf Szabó, AITIA	BME, AITIA
Chapter 23 Energy IoT Monitoring Platform	Bosch, BME	Marco Castangia, Polito Gianvito Urgese, Polito	Politecnico di Torino
Chapter 24 Arrowhead Framework on STM32	Bosch, BME	Sara Bocchio	ST-I
Chapter 25 Keycloak integration	BME	Szvetlin Tanyi	AITIA

28.2 Amendments

No.	Date	Version	Subject of Amendments	Author
1	29.09.2020	0.1	Chapter abou Eclipse Kura and Kapua added	Paolo Azzoni (integrated by Johannes Kristan)
2	30.09.2020	0.2	Some fixes and removals in Chapter 2 Eclipse IoT	Johannes Kristan
3	30.09.2020	0.3	Merged separate References sections to a unified references list	Johannes Kristan
4	30.09.2020	0.4	Chapter about Arrowhead Framework on STM32 added	Sara Bocchio
5	14.10.2020	0.5	Chapter on KeyCloak Integration added	Szvetlin Tanyi
6	16.10.2020	0.6	Chapter on Workflow Choreographer and Executor added	Pal Varga
7	27.10.2020	0.7	Core v4.2-related information added; editing	Pal Varga
8	29.10.2020	0.8	Intro and Conclusion	Pal Varga
9	30.10.2020	0.9	Editing for review	Pal Varga

ARROWHEAD
TOOLS

28.3 Quality assurance

No	Date	Version	Approved by
1	22.11.2020	1.0	Jerker Delsing