Budapesti University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Cyber-physical systems collaborating in a service oriented architecture

Scientific Students' Associations Conference paper

Author
Csaba Hegedűs

Supervisor
Dr. Varga Pál

October 28, 2016

Contents

Kivonat

Az eljövendő Dolgok Internetének világában, ahol globális összekapcsolódásokat és együttműködő rendszereket vízionálnak, minden elosztottan és önszerveződően fog működni. Azonban ez hatalmas kihívásnak minősül, mely olyan megközelítéseket igényel, amelyek messze állnak az elterjedéstől, mert gyakran még nem is léteznek. Az okos eszközök hálózatainak menedzselése akár kisebb nagyságrendben is (nemhogy 50 milliárd eszközre [7]) még mindig jelentős kihívásokat rejtő kutatási terület.

Mindemellett az ipari alkalmazási területek is egyre fontosabbak, ahol flexibilitást és együttműködést várnának el gyakran hagyományos rendszerekre építve. E területen az ipari (például termelési) folyamatok újfajta technológiákkal és architektúrákban való megvalósítása egy igazán nehéz feladat, mivel meg kell őrizni többek között a valós idejű QoS-t és a biztonság minden árnyalatát.

Az Arrowhead projekt [8] egy konzorcium víziója, mely az Európai Unió támogatását is élvezi. Küldetése, hogy keretet adjon az ipari automatizálás területén nem csak a technológiai, hanem a teljes megvalósításban az együttműködés megteremtésére. Öt fő területet céloz meg: ipari termelés és feldolgozóipar, okos városok, elektromobilitás, energiatermelés és a villamos energia piac automatizálása.

Az Arrowhead Keretrendszer célja, hogy megteremtse a megfelelő koncepciókat, tervezési folyamatokat és egy referencia implementációt. Ebbe a szolgáltatás-orientált architektúrába kiberfizikai rendszerek integrálhatók, melyeket központi rendszerek irányítanak és felügyelnek zárt működési környezetben, miközben végzik hagyományos technológiai feladataikat.

E dolgozat a keretrendszer fő központi szolgáltatását tárgyalja, amely orchesztrációs feladatokat lát el e zárt ipari rendszerekben [8]. E folyamat során - kezelve a különböző szintű rendszerképességeket és autonómiákat - bizonyos mértékű központosított vezérélést valósítunk meg a hálózat felett, együttműködve a többi központi rendszerrel, például a Kapuőr modullal [9].

Azért, hogy e központi irányítóval szemben támasztott követelményeknek megfeleljen, számos alapvetést, folyamatot, központi rendszert és szolgáltatást kellett finomítani vagy megtervezni, melyek ez által szintén e munka fókuszába kerülnek.

Abstract

In this upcoming world of the Internet of Things, global interconnectedness and interoperability is promised between any device at any time - all orchestrated run-time in a distributed manner. However, this grand challenge assumes new fundamentals and approaches that are far from existing let alone being proven. Creating functioning networks of smart things even in smaller scale, or for 50 billion devices [21], is still a very active research area.

Nevertheless, industrial applications are also pitching in, flexibility, interoperability and integrability of legacy and new devices is of their high interests. Creating industrial (e.g. manufacturing) processes by utilizing modern concepts and technologies while retaining e.g. QoS, safety or security is also not a minor task.

The Arrowhead Project is the vision of a consortium with the backing of the European Union [23]. It aims nothing less than to create a full-scoped framework enabling collaborative automation by networked embedded devices. It targets use cases from five business domains: production (process and manufacturing), smart buildings and infrastructures, electro mobility, energy production and virtual markets of energy.

The Arrowhead Technological Framework is aimed at providing fundamentals, concepts, guidelines and reference architecture implementations to achieve its goal. This service-oriented architecture builds upon cyber-physical systems, integrating and governing them in closed operational environments while traditional industrial technologies and restrictions are still applied.

This work mainly mainly details my contribution for the design and development of the core services of the framework: orchestration [23]. This core process has to deal with various levels of system autonomy and capabilities while retaining a level of centralized control over the network in co-operation with other core systems of the framework (e.g. with the Gatekeeper module [24]).

To successfully handle the expectations set for this central governance entity, certain fundamentals, processes, core systems and services had to be refined or newly established - which are naturally also in the scope of this work.

Introduction

There are various technology domains - from electro-mobility through smart buildings and infrastructures to energy production - that have great impact on our life. This impact will be even more positive when the interoperability of these systems becomes a reality. There should be some commonly defined patterns regarding the communication and orchestration of these currently independently designed systems. Such patterns will enable standard design methods of complex services provided by functionally heterogeneous System-of-Systems.

This is often referred to as the fourth industrial revolution - or shortly Industry 4.0. The design principles are still a work in progress, but the fundamentals have been laid down [25]:

- Interoperability: The ability of devices, sensors and people to connect via the Internet of Things
- Information transparency: The ability of information systems to create a virtual copy of the physical world by enriching digital plant models.
- Technical assistance: The ability of assistance systems to aggregate and visualize information for advanced and spot-on decision making.
- Decentralized decisions: Cyber-physical systems are ought to operate as autonomously as possible without human intervention.

Accommodating these expectations are not an easy challenge, most parts are still considered green-field research areas - and therefore rewarding ones. Many technologies and philosophies are competing on how to achieve it - one of it is now the Arrowhead.

The Arrowhead project

The Arrowhead Project [13] is the vision of an industrial consortium with the backing of the European Union. It aims nothing less than to create a full-scoped framework enabling collaborative automation by networked embedded devices. The grand challenges it faces are the creation of interoperability and integrability of almost any device. Initiatives like these could solve the ever growing problem of the deep-rooted incompatibilities in the IoT and automation worlds [13].

The ECSEL Arrowhead project started in 2013 with over 80 partners. These stakeholders also show heterogeneity as opening to an integrated IoT world is in the interest of many technological areas. From smart bearings to smart cities, there is the need for interoperability and convergence in the solutions. Since the primary objective is to create a common platform for all sorts of future implementations, the primary pilot (use case) scenarios of the project involve several business domains:

- industrial automation and production: both the processing and manufacturing sectors,
- smart buildings and infrastructure and smart cities,
- electro mobility: smart electric cars, smart traffic control,
- energy sector: virtual market of energy, energy production and end-user services.

The project also covers various activities in order to reach the above mentioned, general targets. These involve (i) creating a cloud-based overall system architecture based on Service Oriented Architecture (SOA) principles [13], (ii) establishing common design and documentation guidelines [5] and (iii) building pilot applications based on its technological framework.

This paper presents an architectural design to solve topical problems by proposing several additions and modifications of the current overall architecture, while respecting the system design principles. Furthermore, it also identifies the advantages and issues that arise with those new concepts. Finally, it presents the reference implementation of this framework created by the author and lays out potential future work.

Outline of this work

In the following chapters, this paper will analyze and augment the Arrowhead Framework. In chapter 1, I will present related works that are necessary to understand the context that are required for latter chapters. Firstly, Service Oriented Architectures (SOA) are discussed as the key design style for this work. Since it is mostly associated with Web Service technologies, they are also discussed (among others) as the primary references for the Arrowhead Framework as a consequence. Moreover, since the main focus of this paper is orchestration, an overview is presented on what orchestration means within various technological domains. After that, the Arrowhead Framework is presented in details: I will characterize the SOA based approach of the project together with the core elements.

In chapter 2, the my main contribution to the framework is discussed. Firstly, the specification is set up based on the requirements and use cases. After that, governance related tasks and processes are discussed for Arrowhead Local Clouds. Further considerations are made for example for security and QoS issues.

Chapter 3 discusses the reference implementation that realizes the architecture described in chapter 2. Conceptual validation and applicability of the framework is presented in the special use case of the electromobility scenario - as showcased at the IEEE IECON'16 conference. Also, preliminary results on the technical assessment of the implemented proof-of-concept is documented.

Finally, chapter 4 will lays the corner stones for future work - since the problem space is much wider than run-time governance of Systems-of-Systems. Moreover, this framework is mostly designed for future additions. It contains placeholders and anchor points for various elements ranging from additional Core Systems to a global configurator engineering tool. In order to move towards the applicability and standardization of this work, clarification of these is inevitable.

Chapter 1

Related work

1.1 Service oriented architectures

Software has become increasingly complex since the early days of computing. Many things tend to hinder interoperability: various stakeholders and their conflicting interests, competing technologies and so on. This leads to monolithic system architectures and extreme measures taken. Even between different versions of the same product (i.e. within an API) we can see exponential growth in complexity and system overhead.

1.1.1 General introduction to SOA

Service oriented architectures (SOA) aim to handle this [20]. There are several definitions and descriptions available, OASIS [38] defines it the following way:

A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

This approach can be interpreted as an extension of the interface concept used in the object-oriented programming paradigm. A SOA based system is built upon a collection of loosely coupled components that are somehow interconnected: they provide services to one another [4]. Fig. 1.1 depicts how this can be abstracted (and presented in the Arrowhead nomenclature).

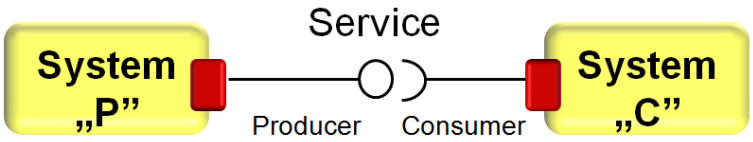

Figure 1.1: *Services produced and consumed by systems*

The building blocks of the SOA systems are these services. There is no standardization on what characteristics services have, or how they can be implemented. The authors of [44] emphasized the following characteristics of service abstraction:

- Loose coupling: hardwired connections between entities are not permitted.
- Rather services are dynamically discoverable upon need (run-time).
- Services are self-contained and modular. A service supports a set of interfaces and these interfaces are logically cohesive (they implement the same functionality).
- Modular understandability: the user has to be able to use the service without having knowledge of any other underlying implementation details.
- Modular decomposability: a complex service can be created from simpler atomic services.
- Interoperability: systems using different platforms and programming languages should able to communicate with each other using services.

SOA is primarily associated with the Web Service stack. However, it is worth noting, that the SOA principles can be implemented using (nearly) any technology - might be even used outside of information technology purposes [15]. Its main advantage lies in its emphasis on a minimalistic common ground and thrive for resource reuse. Moreover, late binding essentially has the capability of providing reconfigurability since service discovery can be tailored to retrieve specially chosen service providers.

1.1.2 Web Services

Web Services (WS) are basically a set of protocols and standards to provide (mainly) a machine-to-machine interoperability layer mainly managed by OASIS [38] and W3C [46]. It alleviates certain hardships that are related to data exchange using web technologies.

This approach is nowadays used nearly everywherey in the ICT domain. Smart phones, the Internet of Things (IoT) and also enterprise applications utilize some sort of web technology. Its unprecedented prevalence and versatility can be mostly attributed to its simple yet elegant transfer protocol stack (HyperText Transfer Protocol - HTTP), security capabilities (e.g. Secure Socket Layer - SSL) and independence of the underlying programming languages or infrastructure.

Representational State Transfer protocols (as HTTP is one) provide further advantageous capabilities. These are then often combined in WS applications and hence their characteristics have also become part of general SOA approaches (among others):

- Stateless: status and context is not stored for the partner (cacheability is a consequence).
- Client-server architecture: communicating partners are separated through a pre-defined interface structure.
- Layered system: a client might not be able to tell whether it is directly connected to an end server (provides scalability via e.g. load balancing).
- Self descriptive messages: resources hold enough information on how to process them.

The Web Services stack implements late binding and service lookup in a particularly interesting way. It is resolved with the introduction of Universal Description, Discovery and Integration (UDDI [48]) servers that are registers storing service descriptions formatted in Web Services Description Language (WSDL). These descriptors can be highly complex (usually formatted in

XML) and they include - among others - the URL and payload structure for requests and a general description on what the service is offering (that is filled out manually). Matchmaking of service lookups to service providers have come a long way from string comparisons: there are now automated WSDL matchmakers that take semantic considerations into account as well since the description parts are manually entered [49].

1.1.3 Service orchestration

Components and services - as per definition - do not create automation on their own. Services are merely a static representation of what a component is capable of. Connecting the components and instructing them on what services to consume (and when) should be tasked with a separate control entity that is called an orchestration. One of the focal points of this paper is considering service orchestration. Therefore, it is viable to survey already existing platforms and approaches and draw conclusions on how to design the capabilities of SOA-based interoperability of the Arrowhead framework. This section explores service orchestration starting from the definitions and then what this term actually means in the various technological domains.

In the WS nomenclature, orchestration refers to a control function that coordinates interactions, flow control and transaction management necessary within a service provider [20]. According to [18] it is a behavior that enables the provider to e.g. fetch or manipulate data structures to provide its service: a composition of sub-services is created that is then offered out as a new service. Meanwhile, choreography refers to a focused collaboration process between service providers and consumers to achieve a certain goal. A choreography also describes when tasks and interactions may happen by describing a general macro-lensed operational flow - without relying on specific instances of the involved types (roles) [ibid.].

Moreover, in Web Services, all of these have their own XML-based description languages defined. The Business Process Execution Language (BPEL) [34] defines abstract processes in a script-like language that is supported by common WS implementations. It also includes objects design to describe error handling scenarios, e.g. when a sub-service fails to deliver. This enables for a detailed orchestration to provide a composition of services as a whole. Regarding choreography, the Web Service Conversation Language (WSCL) [47] is the de facto accepted scripting language for this function – also based on XML.

It is worth noting that there is general orchestration and choreography-related ongoing research since this control theory area is far from being well-established (e.g. [27, 50]). They cover system roles, actions and state representations established in a ontologies. Various operational management layer issues are also of interest: state machines in service providers and methods on rooting out possible deadlock scenarios. Moreover, there are ongoing projects funded by the European Union that aim to create unified process scripting languages suitable to extend general WS tasks outside of the enterprise domain (e.g. Chorevolution [12]). Currently, high level control over a service-oriented establishment is not explored in Arrowhead - however researched as future work.

The Web Services stack is heavily business-oriented and a mature SOA - from ontologies and

theoretical research to enterprise platforms. One of its drawbacks, the WS messages and protocol stacks have a very high computational overhead.

	Orchestration	Security	Protocol stack	Implementations
Web Services	- orchestration: service composition at provider side - choreography: macro-lensed configuration of components for a specific target	- SSL/TLS using X.509 hierarchy - advanced authorization based on roles	-transfer: HTTP -data representation: mainly XML	- Java EE - Microsoft .NET Framework - DWPS
IT Clouds	- configuring cloud resources for service deployment - monitoring of deployment (telemetry) - event and error handling - policy-based orch.	- orchestrators configure security measurements - VPN solutions - Virtual Network Functions run by physical devices	- various - some based on HTTP	- IBM SmartCloud Orchestrator - Intel Ciao - Apex Lake
Software Defined Networking	- coordinate network resources and actives for new services - deployment of network virtual functions - deployment monitoring (QoS)		-various - OpenFlow - OpenStack	
Formalized orchestration languages	- process control-flow description and verification	Not involved	- scripting languages (e.g. Java-based)	- Chor - ChorEvolution

Figure 1.2: *Orchestration in the various technological domains*

Besides WS, service orchestration is also referred to in other technological domains. These include (i) IT clouds [35, 19], (ii) Software Defined Networking [29, 37, 10, 39, 9] and formalized orchestration languages [26, 51, 27, 12, 50, 52] - among others. Table 1.2 concludes their important aspects and table 1.3 identifies useful architectural components and shortcomings of the solutions for this use case.

	Useful concepts	Inadequacies
Web Services	- applicable protocol stack - orchestration and choreography solutions	- large protocol stack overhead - only HTTP and XML - no real-time QoS capabilities
IT Clouds **Software Defined Networking**	- architecture for central governance of deployments - reconfiguration methodologies - performance monitoring, event detection techniques (telemetry)	- Instrumentational toolsets are connected to cloud and network resources - Requires advanced end-point capabilities - suited for infrastructure management
Formalized orchestration languages	-scripting languages - evaluation methodologies	Mainly proof-of-concept

Figure 1.3: *Evaluation of the concepts and solutions*

To conclude, while these technologies are good reference points for the Arrowhead Framework,

they do not substitute the Framework and this work. Parts of them may answer many of the relevant issues and therefore elements, concepts and even implementation can be reused. However, they (one by one) do not cover the complete problem space - a composition of their advantages are welcome however. The list of currently unresolved issues to resolve include the followings:

- Low powered or legacy devices are involved: not capable of the complex protocol stacks and instead legacy automation protocols are involved in most cases. They are also mostly incapable of implementing self-orchestration and execution of advanced choreography on their own.
- Hence some advanced end-point functionalities (restricted autonomy) have to be allocated to other elements in the architecture.
- No cloud technologies are involved.
- Often physically isolated networks are expected to maintain security.
- System-of-Systems model: pre-determined macro-lensed targets drive industrial applications (e.g. production lines), therefore some sort of run-time governance is required based on physical process flows.
- Real-time resource management and network QoS is considered essential, yet traditional QoS provisioning techniques might not be applicable in some cases.

1.1.4 (Industrial) Internet of Things

The Internet of Things paradigm is dealing with somewhat somewhat different issues than traditional WS applications. Nowadays, the emerging number of IoT platforms makes it a laborious task to survey the industry. Various sensing, cloud and stream processing technologies can be (among others) brought together in an arbitrary way to realize certain well defined tasks.

There is common ground among them, however. A survey of 17 well-known platforms (supported by "big players of the industry") [16] has shown that although the goals are overlapping, solutions differ in scope and philosophy.

Widespread elements include the thrive for rapid application development (throughout the product lifecycle) and easy (re-)configurability. Moreover, the potential for scalability and other deployment considerations are also considered important and possible benchmark characteristics. Nevertheless, platforms and frameworks have to recognize and facilitate the followings in order to be successful [ibid.]:

- Enable devices to securely expose interfaces for others to consume.
- Enable systems to retain their implemented application protocols.
- Enable applications to participate in networks accordingly to their capabilities.
- Governance - Enable management and governance of heterogeneous networks of devices.

However, traditional industrial applications differ from this world and IoT platforms are not suited to be fully integrated in there. The emphasis on security and safety limits flexibility and (sometimes) imagination. The strict engineering guidelines, ones based on the factory automation industry standard ISA-95 pyramid [43] have to be respected. A current trend of transitioning

traditional automation architectures of industrial systems into IoT-like approaches is sketched in Fig. 1.4.

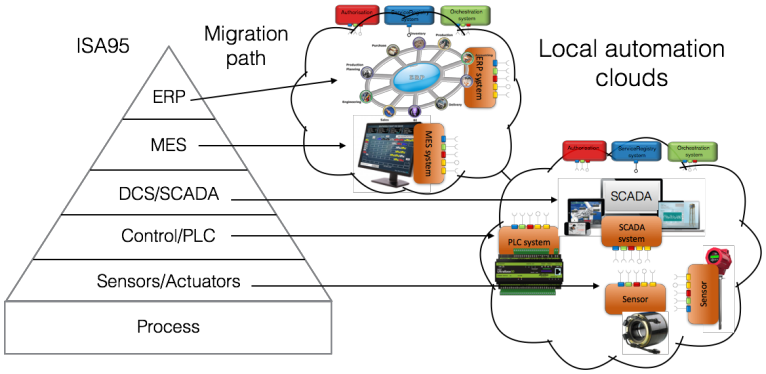

Figure 1.4: *The ISA-95 system hierarchy*

An example on communication security shows the hardships involved. Even though within closed environments (like those within factories) network security is considered an unnecessary overhead on endpoints (that might be a small sensor) as hard real-time delay limits have to be kept end-to-end. Also, no routable protocol might be implemented directly to the outside world since gateways might get compromised - and yet outbound connections are still desired. Such justifiable paranoia has to be supported framework targeting industrial applications.

To conclude, currently available IoT platforms are not designed to operate in the industrial automation domain. Therefore, there is valid demand for more advanced, specialized approaches - like the Arrowhead Framework and this work within.

1.2 The Arrowhead Framework[1]

Having heterogeneous systems (or architectures) work together is not an easy problem to solve - especially in the automation domain. Legacy protocols, commercial off-the-shelf products and monolithic architectures often cripple interoperability and dynamic reconfigurability. This section details the baseline Arrowhead Framework and its fundamental components. This architecture is the take-off point of this work.

For the sake of distinguishing general service oriented objects and those of within Arrowhead, I will use capital letters for Systems, Services and Clouds that are to be understood as Arrowhead objects. This is necessary here, since certain characteristics and definitions of these entities might differ from the mainstream understanding of those terms.

1.2.1 Local Clouds

The Arrowhead Framework uses the approach of Service Oriented Architectures (SOA) to tackle this problem: it aims at providing interoperability by facilitating the service interactions within

[1] This section is based on [45].

closed or at least separated automation environments, called Local Clouds (LC). Hence creating central governance with a minimalistic set of core components that take over certain configuration tasks from individual systems.

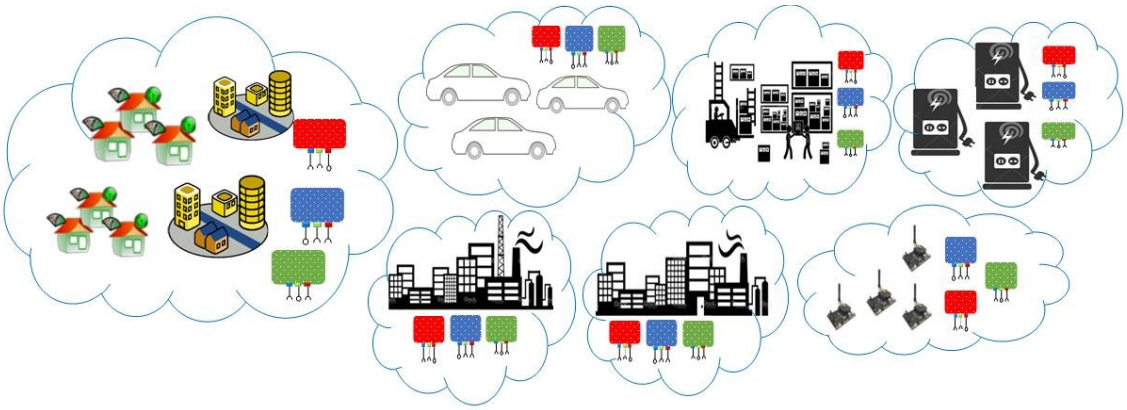

Figure 1.5: *Arrowhead Local Clouds deployed in different domains*

Currently, there is no strict boundaries on what might become a Local Cloud.These LC's might fulfill various tasks and can have their own sets of appointed stakeholders (e.g. their operators or developers). However, they all have their operational boundaries, let those be functional, geographical or network-segmented. Nevertheless, they must be governed through their own instances of the Arrowhead Core Systems, as Fig. 1.5 suggests. They are clouds in the sense that they use common resources: the Core Systems of that domain. These common Core System resources (e.g. related to Service Registry, Authorization or Orchestration) are used by all kinds of other entities - applications - in the network, and can also be implemented in a distributed way.

1.2.2 Systems and Services

In such Arrowhead Local Clouds there can be an arbitrary number of Systems that can provide and consume Services from one another: they create and finish *servicing instances* dynamically in run-time. Figure 1.1 depicts these relations. Services are defined so that loose-coupling, late binding, and service discoverability can be realized.

In this context, Arrowhead-compatible Systems are not just service endpoints, but can be realized by a wide range of devices: from a small temperature sensor, in certain use cases, up to very complex cyber-physical systems of a production plant. A resource can be a temperature sensor or a power consumption meter. In this sense, a Service is connected to the temperature itself regardless of the access method or technology (interface).

All Arrowhead compliant Systems will also be capable of consuming different types of Services in their different implementations - that they might need and compatible with - during runtime. Services are also not bounded by System instances, since a Service can be realized in an arbitrary number of Service Producers and Service Consumers.

Arrowhead-compatible Systems must use the mandatory Core Systems and their Core Services provided by the Framework to realize their operational targets (as shown in Figure 1.6): to communicate with other Systems using dedicated application protocols.

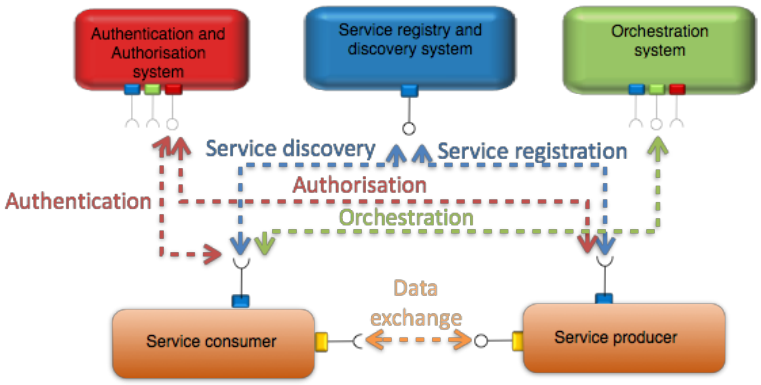

Figure 1.6: *Arrowhead Core Systems*

1.2.3 Mandatory Core Systems and their Services

In accordance with [45], this work attributes the same functionality to each of the Core Systems. There are three mandatory Core Systems.

The *Service Registry* stores all the Systems (that are currently available in the network) and their service offerings. Systems have to announce their presence, and the services they can offer. The registry takes note of this information when systems come on-line, and might have to revoke them when they go off-line. This System is implemented using DNS (Domain Name Service) based Service Discovery protocol dedicated to resource management in a network [11].It uses an (string) identifier for the Service implementation and the Systems running it.

For a service request, the server provides information about suitable network nodes by using SRV (service), TXT (text) and PTR (pointer) records. The hosts offering services have to publish details of their available services: instance, service type, domain name and optional configuration parameters [ibid.].

The *Authorization System* - as its name suggests - manages authentication and authorization (AA) tasks, however, it covers some other security-related issues as well (e.g. certificate handling). This System is not clearly defined in the baseline framework. Advanced security measurements are required - that is in scope of this work.

The *Orchestrator* is responsible for instrumenting each System in the cloud: where to connect and what to consume. It instructs Systems so by pointing towards specific Service Providers to consume specific Service(s) from. This has to be done by a simple request-response sequence, which ends with the requester System receiving a Service endpoint. After these, the System is obligated to consume from that Service instance. In the baseline architecture, this System is configured manually with simplistic lists of hardwired "Orchestration rules" strings.

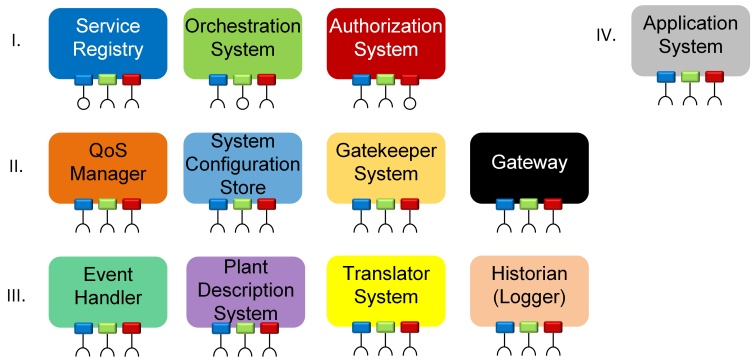

Figure 1.7: *Arrowhead Core Systems in scope*

1.2.4 Supporting Core Systems and their Services

Since the definition of the mandatory Core Systems, there are now further automation supporting Core Systems available. These are currently integrated by this work (II.) or are in the scope of future endeavors to be integrated in the Framework (III.), as Figure 1.7 suggests.

The QoS Manager concept aims to make the real-time constrained embedded devices integrable in a Local Cloud (i.e. actuators) [22]. An other invaluable aspect of such an automation support framework is the centralized management of resources. Resources in a Local Cloud can vary from e.g. processor task time slots of embedded Systems through real-time networking QoS up to setting client-server servicing requirements (e.g. transactions per second). This resource management is tasked with the QoS Manager [22]. Its two major processes are (i) the verification of the requirements (whether the resource reservation can be made), and (ii) the actual reservation process. The QoS Monitor sub-module is responsible of monitoring whether SLA-s are kept or intervention is required.

The System Configuration Store [7] provides customized storage for firmware and configuration file updates for Application Systems. This storage solution has to support a multitude of transport and firmware management protocols in a secure and safe way. It is not an easy task, even in closed environments, to manage the identification of the devices. The other solution (besides having the whole firmware downloaded), is to define a unified run-time Arrowhead configuration layer (an API) that will then be required by Application Systems to possess. However, this concept is currently not applicable.

The Plant Description system [8] aims to integrate various topologies that might be used to describe the same establishment (e.g. one based on physical layout and others based on process-oriented groupings), as seen in fig. 1.8. This engineering tool gives a simplified solution for describing the different hierarchies and topologies that may exist within the same plant or system of systems, sometimes defined according to different standards or procedures. The Plant Description aims to provide a basic common data structure which can be used to refer to different objects in a large system or system-of-systems and the relations between the objects. This System and its possible future use will be examined later on.

The Historian hosts application data for Systems, and stores the data in query-able form. The

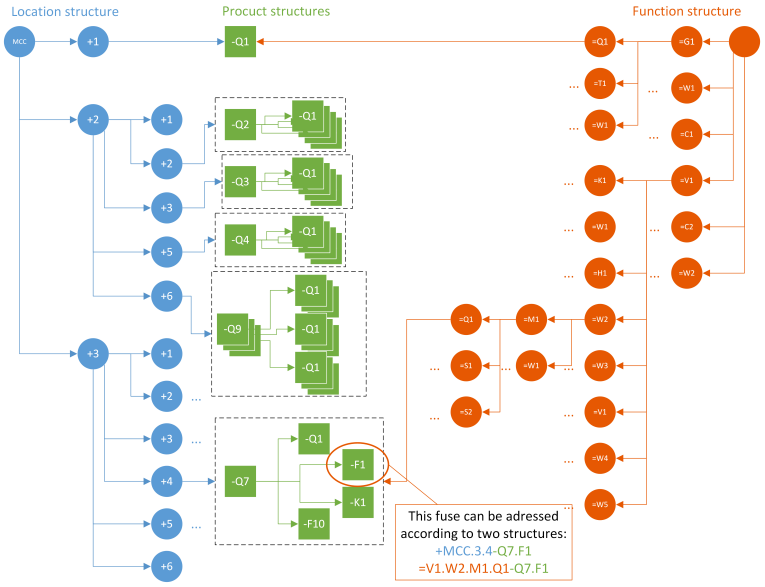

Figure 1.8: *Arrowhead Core Systems in scope*

Translator System [17] can (will) provide application-level translation so that a producer and consumer implementing a different protocol of the same Service can still interact. Event handlers are tasked with the run-time detection of various events and errors and triggering the appropriate Systems (in a publish - subscribe manner).

1.2.5 Inter-Cloud servicing in Arrowhead[2]

Arrowhead Local Clouds might be sufficient to handle certain tasks in their isolation. However, as one single Arrowhead Cloud (one SoS) cannot serve for all, there is need for inter-cloud operations to achieve true global interconnectivity within the Framework.

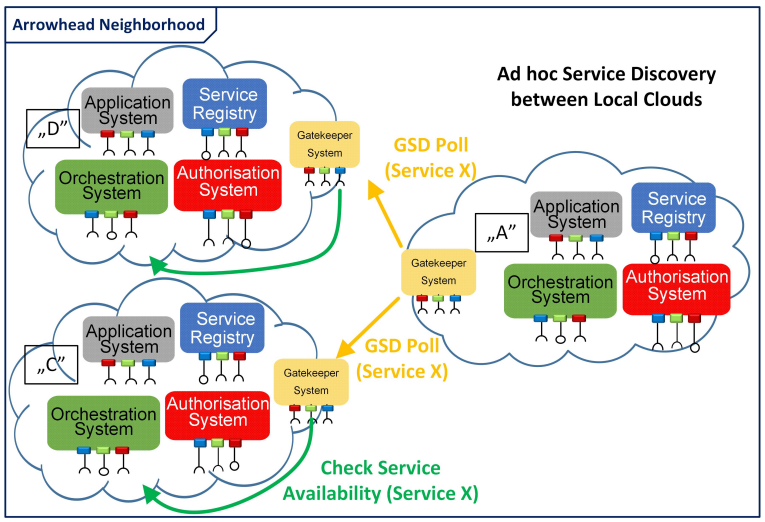

Figure 1.9: *Global Service Discovery between Local Clouds*

[2]This subsection is based on my article [24] and Scientific Student' Associations Conference Paper of 2015.

To this end, an inter-Cloud servicing architecture was introduced in [24] based on two, new automation supporting Core Systems: the Gatekeeper and the Network Manager (a gateway entity). This architecture builds heavily upon the Core Systems' exclusive involvement in facilitating and managing the connections (servicing instances) within their authority: in this sense, the Core Systems are responsible for all other Application Systems and their data handled in the Local Cloud. The Gatekeeper provides essentially two services for the mandatory Core Systems. The first is the Global Service Discovery (GSD) process, which aims at locating adequate service offerings in neighboring Clouds (as Fig. 1.9 suggests). The second is the Inter-Cloud Negotiations (ICN) process, in which mutual trust is established between two Clouds and the actual connection between endpoints is then built up.

After these processes, the Network Manager is responsible for creating the data path between the Service Provider and the Consumer. We assumed here that the Core Systems are configured properly, so that they can securely establish connections with other Clouds using the services of the Gatekeeper. This setup includes some security considerations taking place, e.g. declaring inter-Cloud access rights (from and to other Local Clouds) in the Authorization Systems, or setting up the trusted neighborhood domains.

Chapter 2

Governance of Local Clouds

This chapter refine the architecture of the Arrowhead Framework. Firstly, it collects and analyzes expectations and sets up specification. Based on that, new data structures are introduced ("common descriptor objects") to be used in the new Core Systems.

The main contribution is here, however, the introduction of an advanced and customizable orchestration schema of a Local Cloud that integrates various processes and hence provides the first line of centralized governance. Here, the definition for orchestration used is based on the baseline framework, as discussed in section 1.2.

Moreover, the introduced orchestration process has to have focus on two main aspects (as discussed previously), regarding security and QoS. This architecture builds heavily upon the Core Systems' exclusive involvement in facilitating and managing the connections (servicing instances) within their authority: in this sense, the Core Systems are responsible for all other Application Systems and their data handled in the Local Cloud.

2.1 Requirements and expectations

In its evolution, Generation 1 of the Arrowhead Framework provided the fundamental principles and preliminary implementation for the mandatory Core Services - whereas Generation 2 provided further Core Services and their mature implementations. Generation 3 introduced automation support Core Systems - however as separately developed modules, yet to be integrated whole.

2.1.1 Scope of functionality

The aim of this subsection is to re-establish Core System capabilities, tasks and responsibilities in order to facilitate the following identified expectations towards the evolution of the Framework:

1. advanced authentication and authorization capabilities,
2. dynamic service orchestration process,

3. real-time communication (QoS) and resource allocation functionalities,
4. event and alarm handling capabilities,
5. the integration of automation support services developed in the project separately (such as the Historian),
6. integration of the Service protocol translation capability [17],
7. inter-Cloud orchestration,
8. secure data path among Local Clouds.

Security (1) is discussed in section 2.3.2 and its integration within the orchestration process. It is also connected to securing the data path inbetween Local Clouds(8). The chosen solution is based on the X.509 certificate hierarchy as discussed in section 2.3.2.

Moreover, in some use case scenarios a dynamical orchestration is more suitable than a static database: a matchmaker of some sorts. It has to be able to (2) dynamically (in run-time) allocate Service Providers to Systems seeking Services based on the current status of the SoS. This is leading to a complex orchestration process and interfaces.

2.1.2 Levels of Application System autonomy

The Core Systems need to support a multitude of Application Systems from "dumb" sensor nodes up to fully autonomous enterprise systems (e.i. Electronic Resource Planning - ERP). Therefore, the Core System interfaces and security measurements have to be flexible enough to support various levels of Application System autonomy.

Based on the primary use case requirements, the following levels of System autonomy should be considered by any general SOA-based IoT solution – based on the given System's capabilities:

1. *Passive Service Providers*: Such Application Systems only offer certain services. They only have to register these in the Service Registry and respond to inbound connections. For example, "dumb" sensors, that can provide readouts.
2. *Statically pre-configured Service Consumers*: In certain cases, an Application System can suffice its purpose by periodically consuming the same set of Services from hardwired a set of Service Providers. This can happen e.g. in data logging devices that read out and store measurements.
3. *Service Consumers with statically configured Service Providers*: they consume Services autonomously in run-time (according to their implemented operational logic), but they can only consume those certain Services from a hardwired lists of possible Service Providers. This is the case e.g. for processing units that can only access certain sensors or their backups (for example actuating can only happen based on local temperature sensors - but if one fails, there has to be an auxiliary one).
4. *Fully autonomous Service Consumers*: they only require advanced matchmaking if they are seeking Service Providers - their run-time operations are based on autonomous decisions. This matchmaker orchestration process only has to return with an appropriate (suitable) Provider and make resource reservations (and run other centralized tasks if necessary). This is required in a market-like environment, where (e.g. energy) offers have to meet demands.

5. *Self-orchestrating Systems*: they implement all Core Service interfaces on their own and capable of self-orchestration using the appropriate processes. However, centralized resource management might still be required.
6. *System-of-System Configurators*: these Systems (or human operators through HMI) are capable of configuring the Core Systems themselves. Therefore, they stand above run-time operations and can change that.

These Application Systems might co-exist in the same Local Cloud and might even interact with each other. Also, they might not even share the same set of stakeholders and therefore advanced AAA (Authentication - Authorization - Accounting) functionality is required throughout their interactions. Nevertheless, this semi-centralized AAA solution has to be implemented in all levels (even on resource-constrained devices).

2.2 Local Clouds, Systems and Services[1]

One of the new fundamentals here is the strict, forced and logical separation between the Systems and Services abstractions. This conforms with SOA and DNS-SD [11], and unifies the data structures in the Core Systems. Nevertheless, this has not been settled explicitly in the Arrowhead Framework previously: only a non-compulsory naming convention was introduced within the Service Registry (DNS-SD).

Therefore, a new System and a new Service hierarchy has to be introduced to support (not just inter-Cloud) interoperability and matchmaking, as shown in Figure 2.1. The Service hierarchy is reflected by the documentation structure of Arrowhead [5]: Services are identified with a name (declared in the Service Definition document). This name is the has to be unique in its (use-case specific) Service Group. This Service abstraction only defines the functionality provided, but a Service can be implemented many ways using appropriate application protocols (e.g. by SOAP, COAP or RESTful Web Services - WS). A Service Provider can therefore even implement multiple *interfaces* of the same Service. Therefore, Systems do not own Services, merely implement one or a multitude of its interfaces (either the Provider or the Consumer side).

Meanwhile, the unified Cloud and System hierarchy follows a general, process-oriented principle. Local Clouds are bound to have an operator, and an operator might maintain several dedicated clouds depending on its use case. Similarly, Systems are bound to be grouped together logically even within their Local Clouds, e.g. if they are located in the same production line. There is no explicit naming convention defined here, since use cases might have completely different hierarchy between their Systems. However, as a proof-of-concept, the framework implements these in the common descriptor objects used in the Core Systems.

Metadata also needs to be associated with Services (for every instance separately). By design, currently key-value pairs can be added to describe them (e.g. "unit"="°C") embedded in JSON. This will enable the integration of standardized data descriptors, e.g. SenML [33]or IPSO Smart

[1]This section is based on my article regarding overall architecture [23].

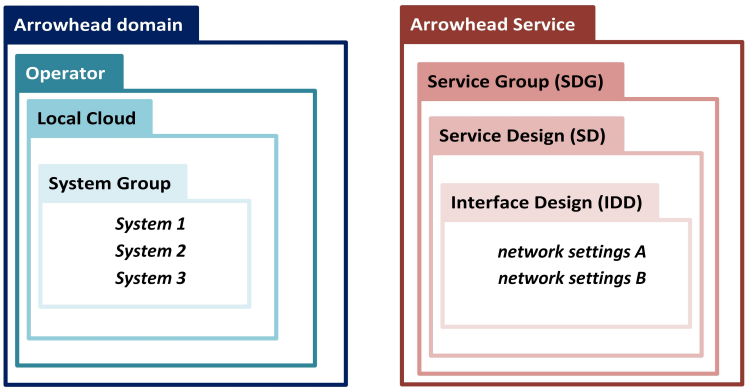

Figure 2.1: *Systems and Services defined for Arrowhead*

Objects [2]. It could also integrate use case specific data ontologies (i.e. MIMOSA for describing maintenance related communications [14]), and can fit into other interoperability strategies, such as the Interoperability Repository System described in [36].

To understand the implied use case specificity of the naming convention, let us consider temperature and humidity sensors in a refrigerator warehouse. They measure the indoor temperature at different locations and heights. Each sector has two sensors that measure the floor- and ceiling-level temperatures. Figure 2.2 shows how these sensors are represented as Arrowhead Systems and how their measurement readout resources can be abstracted as Arrowhead Services.

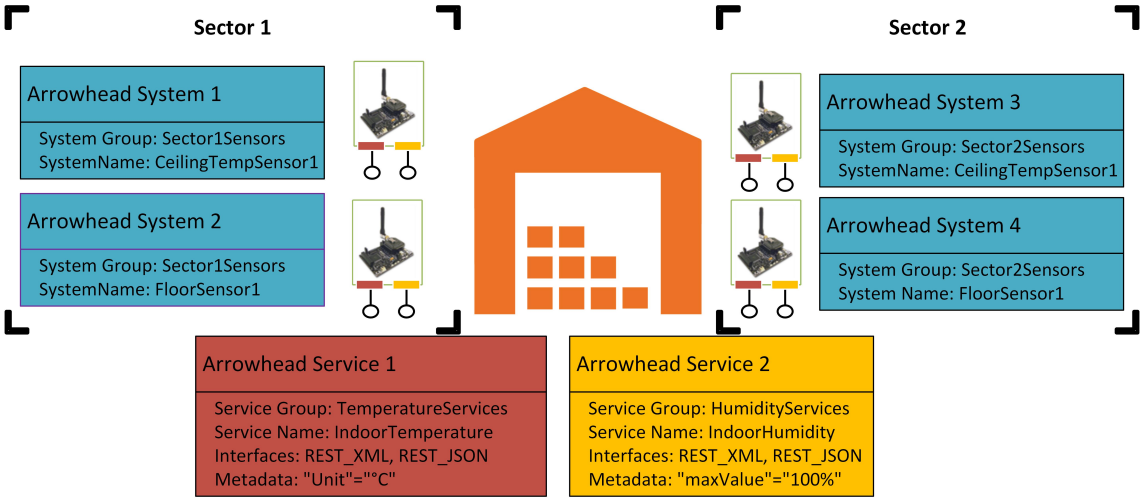

Figure 2.2: *Example for unique identifiers regarding the various Arrowhead elements*

The common descriptor data structures are created to facilitate communications with and between Core Systems. Figure 2.3 describes the classes (objects) involved. However, it is worth noting that this puts certain restrains on what an Arrowhead System can be: an network address paired with a port number where the REST resources (Arrowhead Services) are available.

The drawbacks of this approach is the indeed the limiting effect. A physical system might have multiple network interfaces and its Services might be scattered between multiple ports as well. From a resource management point of view, therefore, another abstraction layer is required on

the devices level since multiple Arrowhead Systems can share the same computational resources. However, this leads outside of the scope of the current work.

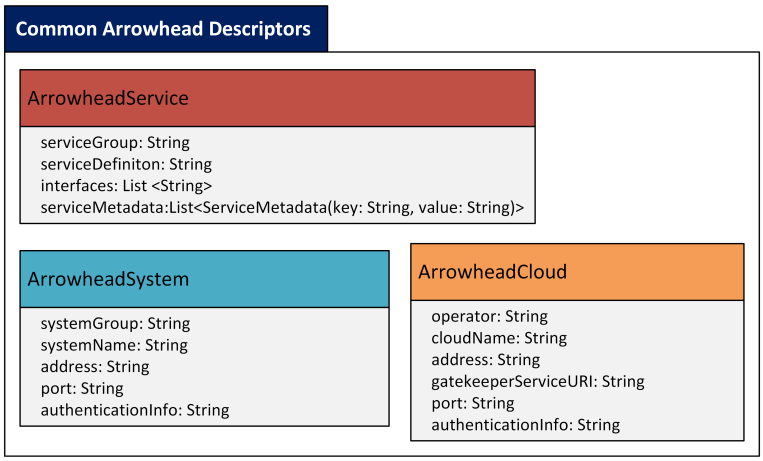

Figure 2.3: *Unique identifiers for the various Arrowhead elements*

2.3 Mandatory Core Systems

2.3.1 Service Registry

The Service Registry stores all the Systems (that are currently available in the network) and their service offerings. Systems have to announce their presence, and the services they can offer. The registry takes note of this information when systems come online, and might have to revoke them when they go offline.

In its earlier generations, it was only accessible through standard DNS protocol. It is a valid platform to store short entries in a domain-based structure. It (i) is highly robust and scalable, (ii) is used across globe as a key technology of the web, (iii) has a lightweight UDP access protocol and (iv) requires nearly zero maintenance if properly configured. However, it has several limitations as well that limits its usability as the sole technology for a Service Registry:

- The entry structure requires workarounds to fit customized data sets (e.g. key-value pairs to described Service metadata).
- It does not provide advanced querying capabilities: it is limited to listing subdomains instead of advanced filtering (based on e.g. Service metadata).
- The entry has size limitation - cannot fit multi-purposed data associated with a Service Provider.
- The access protocol is handed over to TCP when the UDP packet size is overstepped (e.g when a new Service Provider is registering).
- The DNS-SD server does not check whether the pointer is still valid or not (only a Time To Live - TTL value is processed).

Therefore, augmentation of this System is necessary for an advanced matchmaker-typed Orchestrator unit. This is solved through a REST-based Service Registry bridge that stores data in

the DNS server - hence leaves the opportunity for low-powered devices to register through UDP - but provides all the filtering and management capabilities that are missing. This System is implemented as part of this Framework and also uses the common descriptor objects described in the previous section. It is now capable mending the shortcomings by (i) a periodical clean-up of the database by pinging whether the Service Provider is alive, (ii) providing a REST based facade to register and revoke Service offerings (in harmony with other data structures used in the Framework), and (iii) provides an advanced query interface towards the Orchestrator.

2.3.2 Security aspects[2]

In an Arrowhead automation cloud environment, communications can be subject to an extensive amount of threats. These include i.e. spoofing, tampering or Denial of Service attacks, and can compromise the security and integrity of the whole infrastructure. These System-of-Systems still possess a general infrastructural vulnerability, despite its decentralized architecture.

Yet these functionalities are indispensable, considering the involved cyber-physical Systems and business processes. Furthermore, the framework targets the collaboration and interoperability of embedded, and often resource-constrained devices using proprietary or industrial protocols. In many cases the devices have limited capability of performing security tasks such as advanced encryption and decryption utilized by modern secure transmission protocols such as Transport Layer Security (TLS) or the Internet Protocol Security (IPSec).

For such cases, a ticketing-based approach has been developed in [42], which works with Constrained Application Protocol (CoAP). For Message Queuing Telementry Protocol (MQTT) and REST - which use TCP -, security based on TLS would be desirable. It uses a centrally issued ticket as a token of identity but does not contain any identity information directly. To authenticate Application Systems within a Local Cloud one must contact the AA server to verify its identity and privileges. Besides its protocol-restricted implementation (supporting merely CoAP), the weak points of this authentication method make it vulnerable to a number of attacks. Firstly, the challenge-response nature on an insecure channel implies that the ticket itself does not verify the authenticity of the sender - and can be exploited with a man in the middle approach. Secondly, since the service provider has to request the local ticketing AA server to validate the ticket (and therefore authenticate the consumer), it is also easy to bypass this control loop. Thirdly, this validation process is requested at every inbound connection and therefore the AA server is vulnerable to a Denial of Service attacks.

On the other hand, this approach is advantageous for a number of reasons. The service providers are relieved from the processing burden of identifying and verifying the consumers: it is done by the central AA entity. It also bears the future capability of integrating it with admission control functions: service providers can assert whether the inbound connection is ratified by the Core Systems - and can be serviced - or not.

[2]This subsection is based on my article on the security measures [41].

The central governance of Arrowhead has to be able to provide all these capabilities - even the ones that are currently missing. Firstly, Application Systems must have a verifiable identity that cannot be stolen or copied through spoofing and Man-in-the-Middle attacks. Secondly, Service Providers should be able to verify whether inbound connections are sanctioned by the Core Systems or tampering attack is happening. Moreover, trust zones have to be established - in accordance with the Local Cloud concept. Admission control management and achieving accountability is also of interest.

The Public Key Infrastructure (PKI) [28] defines how to create certificates that can be used to achieve authentication, message integrity and confidentiality. PKI builds on Public-key cryptography (PKC), which employs an asymmetric encryption scheme. This means that it uses different keys for the encryption and the decryption processes, compared to symmetric key encryption methods, where the same key is used for the encryption and decryption.

Based on this, an application level AA architecture based on this X.509 PKI infrastructure is proposed here. Each capable Arrowhead System should be provided with a certificate and should therefore have its identity binded to its private key. Every Local Cloud has a Certificate Authority that issues and signs these System certificates. This CA is the root of the trust chain within its Local Cloud and has its own certificate signed by a parent CA. This entity also possesses the Certificate Revocation List defined in the standard: certificates that have been invalidated and therefore not to be accepted.

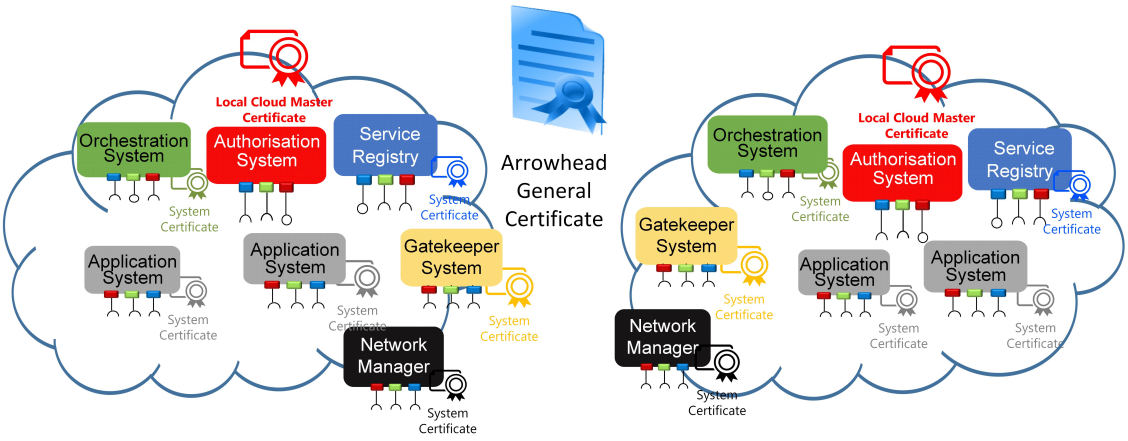

Figure 2.4: *Arrowhead certificate hierarchy*

This chain of trust model fits well into to System hierarchy concept of the Arrowhead framework, as depicted in Fig. 2.4. A general, master Arrowhead certificate can be signed by a well-known trusted CA (such as Comodo or GlobalSign) and issued to the Arrowhead domain owner (e.g. the project consortia). This administrator entity then can issue and sign Local Cloud certificates for operators in its own application process for establishing new Local Clouds. Within these new Clouds then the Authorization System realizes the CA tasks and owns the cloud certificate.

If an Application System in a Local Cloud requires a certificate (e.g. during its deployment procedure), it will have to generate a private-public key pair and submit a Certificate Signing Request (CSR) containing the pair and its identity. There are other fields in a CSR used to verify

```
Subject: C=HU, L=Budapest, O=Manufacturer1,
OU=Fleetcloud1,
CN=TempSensor1.Car1.FleetCloud1.Manufacturer1.arrowhead.eu
```

Figure 2.5: *Identity and hierarchical information stored in certificates*

the identity of the requester. This way an Arrowhead System, bootstrapping procedure can be created, and augmented with certificate generation.

Moreover, Systems acting as Service Providers should also be able to verify that inbound servicing requests are properly authorized and verified by the Core Systems. To this end, we introduce an authorization token building on the ticketing schema that will provide application-level security. A such token is only valid for *one servicing instance*: one Service Consumer is authorized one-time to consume a specific Service from the Service Provider at hand. This information (Consumer - Service - Provider) can be stored in a string and should be decryptable and parseable by Application Systems that require such advanced admission control functionalities. These tokens have the following characteristics:

- It builds on the certificate hierarchy (trust zones);
- Generated by the Authorization System based on access rights;
- This token is then passed on to the Consumer by the Orchestrator during the orchestration process;
- It is only decryptable by the Service Provider and its private certificate key;
- It assures that proper orchestration took place and the Consumer is verified to access the Service;
- Based on this, the Service Provider can either accept or reject the connection.

Here, a certificate structure can be implemented with the above discussed hierarchy, as depicted on Fig. 2.5. This format is a customization of the general X.509 certificate and it bounds the identity of the system specifically to a Local Cloud (which makes spoofing attacks more difficult).

Moreover, besides the Certificate Authority tasks, the Authorization System stores the admission control databases. Currently, these are static tables that contain (Service Consumer - Service - Service Provider) trios for intra-Cloud and (Service - Cloud) consumption rights for inter-Cloud authorization purposes.

Regarding the authorization token itself, it is a mere "SHA1 with RSA" crypted byte stream. Since the used X.509 certificates consist of 2048 bytes, the token itself is approx. 240 bytes - that can be decrypted in one step. This decryption is implementable by embedded devices, and some might even have dedicated hardware acceleration for this task. To verify the issuer CA (the Authorization System of that Local Cloud), a signature is generated for each token.

Fig. 2.6 details what the token includes. With this information, the Provider should be able to verify using the consumer's CN field whether it should accept the connection or reject. This handshake requires a new interface - hence an admission control service has been defined. In case the used strings should overstep the token size limit (using ASCII), the token

```
{
c=''TempSensor1.Car1.FleetCloud1.Manufacturer1'', //consumer CN
s=''RESTJSON.IndoorTemp.TempServices'', //service interface
i=''1477138884'', //Epoch timestamp of issue
to=''1000'' //validity period in ms
}
```

Figure 2.6: *An example authorization token*

2.4 The Orchestrator

The Orchestrator is responsible for instrumenting each System in the Cloud: where to connect and what to consume. It instructs Systems so by pointing towards specific Service Providers to consume specific Service(s) from. This has to be done by a simple request-response sequence, which ends with the requester System receiving a Service endpoint. After these, the System is obligated to consume from that Service instance.

However, this can be done in a multitude of ways. Currently, this work focuses on three major scenarios that have to be resolved by this one Service. The following sections detail these and 2.4.4 presents how this is solved within a single request-response communication:

- Default configuration: System initialization,
- Store-based orchestration: fall back to backup Service Providers,
- Intra-Cloud dynamical orchestration: matchmaking based on the request and
- Inter-Cloud orchestration.

2.4.1 Default configuration

When an Application System wakes up, it will have to request a default orchestration set that will instruct it where to connect after or during initialization. It is of essence here, that further instructions are passed on to the System (e.g. initialization parameters) and that the Service connections are to be made sequentially.

Figure 2.7 shows a generic example. To give a more specific example, we can assume that the first connection (Service X) is made to the System Configuration Store, where the System fetches its operational firmware. Secondly, it will have to register the Services that it provides in the Service Registry (Service Registry) represented as Service Y. Thirdly, it will have to sign up to the Event Handler by consuming Service Z.

2.4.2 Store-based orchestration

There might be operational limits of how Application Systems can be orchestrated, i.e. certain hard-wired service interactions cannot change at all, supposing that an actuator can only access certain sensors. The Arrowhead Framework has to abide by such implementational constraints that shape this world of the industrial Internet of Things.

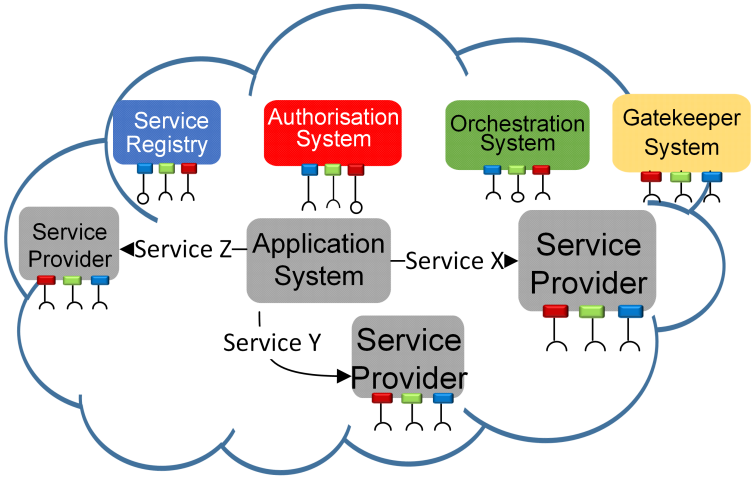

Figure 2.7: *Serving default configuration*

Therefore, when Systems request orchestration, we have to consider if there is a set of orchestration rules defined for it (Consumer - Service combination) in the Orchestration Store. By design, it should support auxiliary Service Providers - if the primary one fails. This enables quick error handling: re-orchestration of a failed servicing instance will return one of the backup data sources. Figure 2.8 describes this scenario.

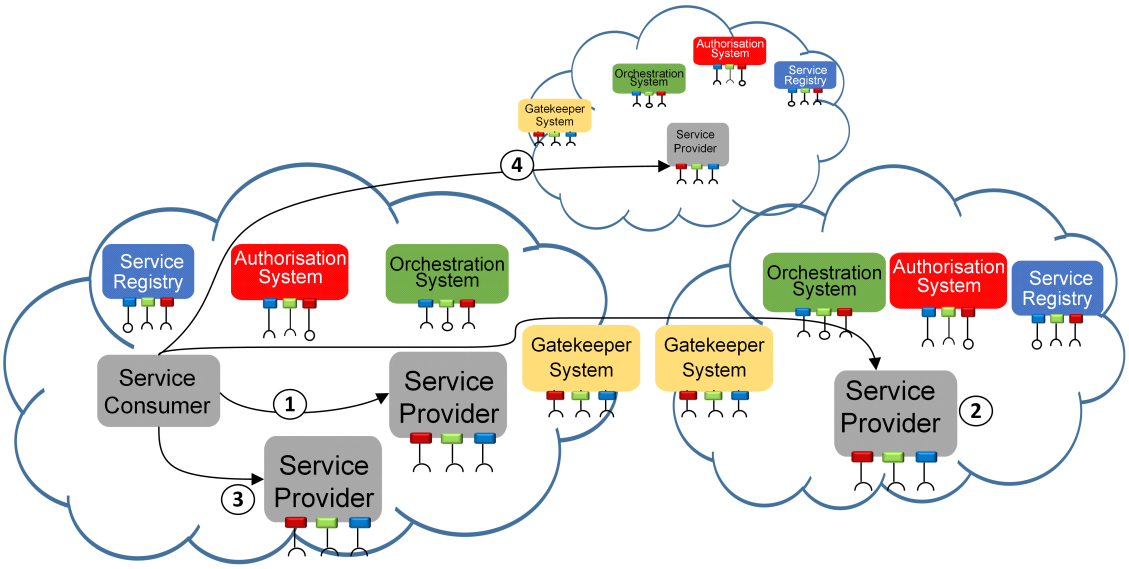

Figure 2.8: *Store-based orchestration*

As Fig. 2.8 suggests, Providers can reside in various Clouds, since Store entries can describe Service Providers in different Clouds as well. If this is the case, then the Gatekeeper's Services are automatically invoked and a servicing connection will be negotiated whether it is possible at that moment. As a consequence the Consumer might get a Service Provider from another Cloud - completely transparently from its point of view.

2.4.3 Dynamical orchestration

The most interesting, flexible Service orchestration scenario is where the dynamical aspects of the System-of-Systems is taken into account. The Orchestrator is the primary decision-maker here that is aware of the current conditions in the SoS. Its primary task is to allocate Service Providers to the Service Requests sent in by Systems. During this *orchestration process* the Orchestrator consults with the other Core Systems and makes a decision based on the responses, as Figure 2.9 suggests. This orchestration process consists of the following:

1. Fetching a list of suitable on-line Service Providers from the Service Registry that offer the Service at hand;
2. Filtering this list of possible Service Providers based on the authorization status and preferences of the Service Consumer;
3. Further filtering this list to find out which Service Provider(s) can satisfy the QoS expectations;
4. Choosing one Service Provider from this filtered list (intra-Cloud matchmaking);
5. Then making the QoS resource reservations for this optimal (Provider - Service - Consumer) combination if availabe;
6. Extending the Service discovery to other Clouds using the Gatekeeper if necessary based on various factors (e.g. local Service Providers are inadequate);
7. Negotiating servicing aspects with a chosen partnering Cloud;
8. Responding to the requester System with an Orchestration Form.

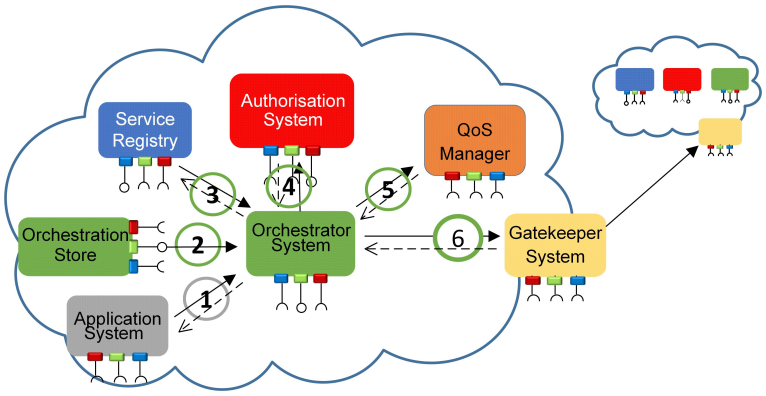

Figure 2.9: *Dynamical matchmaker Orchestrator*

In this dynamical orchestration process, Service Consumers can also have preferences: where they wish to connect to. Although these Systems might be fully aware their operational targets, they still need to request orchestration, since resource reservation and generation of the authorization token is still required - and that can only be done by the Core Systems.

Moreover, the current architecture enables the customization of the matchmaking function. Since various use cases might weight various factors of Service Providers differently (e.g. based on System and Service metadata), it is feasible to provide a placeholder function for this sub-process. The output is one or more suitable Service Providers that match the criteria the best.

To conclude, this matchmaker orchestration process is similar to the UDDI concept from the WS stack - where complex service requests are matchmade. In here, orchestration is a centralized capability and orchestration rules (if there are any) are distillated from overall SoS choreography (currently only manually). This way, "dumb" Application Systems can request new run-time configuration and orchestration upon a central signal distributed through e.g. the Event Handler System.

2.4.4 The internal workings of the Orchestrator

There are two main parts of this System: the Orchestration Store and the Orchestration Service. The Orchestration Store is a statically configured database that contains hardwired orchestration data for Systems in the Cloud. Operators can set up relations between Systems that cannot change runtime, and have to be respected in the orchestration process. One Orchestration Store entry (Fig. 2.10) consists of a Service Provider and a Service specification that must be enforced if the Service Consumer at question requests orchestration. These entries are set by the operators and developers of a Local Cloud suiting the physical processes and restrictions in the the Cloud. The first two orchestration scenarios utilize this database.

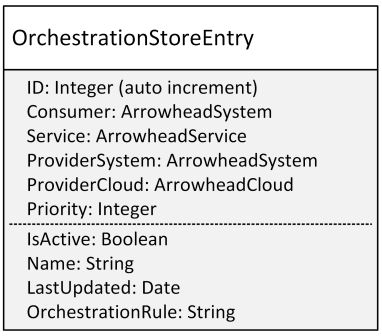

Figure 2.10: *Data structure of the Orchestration Store*

The main interface of this System is towards Application Systems. Fig. 2.11 depicts the invoking and response messages. Systems can submit a Service Request Form (SRF) that has mandatory and optional fields (so that resource-constrained devices don't have to store the whole message template). This message is encoded in JSON by default (or XML) and expresses what the Consumer is looking for. It includes the following:

- *RequesterSystem*: This declares the requester Consumer. This information is also present in its X.509 certificate CN when using SSL.
- *RequestedService*: The Service that the Consumer is looking for. It might be optional in some cases, e.g. when default configuration set is requested.
- *RequestedQoS*: A complex object for QoS expectations set by the Consumer for the servicing.
- *OrchestrationFlags*: These flags specify what is requested and how. They effect the orchestration process.
- *PreferredClouds and Providers*: These fields specify the preferences of the Consumer: where it wants to connect to. It is a priority list, but local Providers are always processed first.

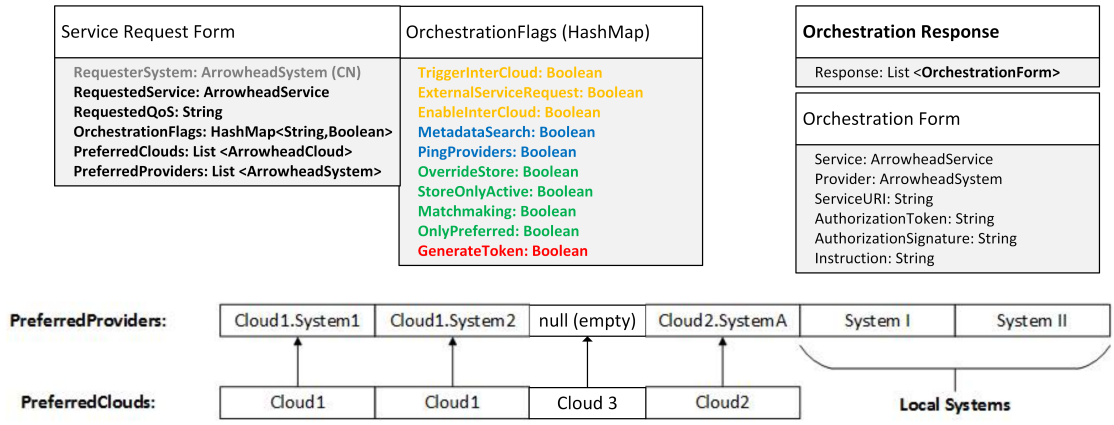

Figure 2.11: *The Service Request Form and Orchestration Response*

After the orchestration process (Fig. 2.12 and detailed in Appendix F.0.6) finishes, the Orchestration Response is returned. It includes the following information:

- *Service*: The exact Service that has to be consumed (defined up to the interface level and metadata description). This stems from the chosen Provider (from its entry in the Service Registry).
- *Provider*: The Service Provider detailed.
- *ServiceURI*: The URL path to be accessed.
- *AuthorizationToken*: This string contains the encrypted token.
- *AuthorizationSignature*: This string verifies that the local Authorization System generated the token.
- *Instructions*: This string is used to pass on further parameters.

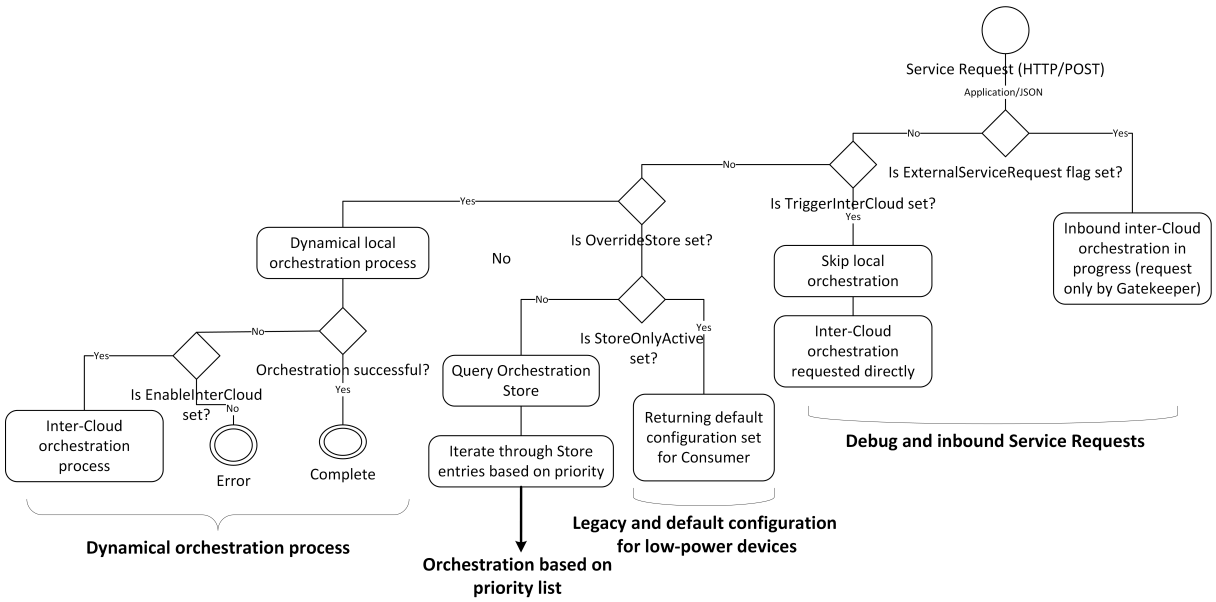

Figure 2.12: *Overview on the orchestration process*

As shown in Fig. 2.11, the Service Consumer can configure the orchestration process it is requesting by setting flags in the Service Request Form. This way, only one payload has to stored in

Application Systems and only filled out in a specific way to trigger different orchestration modes. The simplified decision tree of the Orchestrator is depicted in Fig. 2.12 and the full version in Appendix F.0.6. Since this process is stateless, the Orchestrator itself is stateless. Every resource is processed independently and therefore this REST service could be put behind advanced load balancing logic - and the bottleneck of the core architecture will rely on the other Core Systems. The interactions between Core Systems during orchestration are depicted on Appendices F.0.5 for the intra- and F.0.6 for inter-Cloud scenarios as a whole.

Chapter 3

Verification of the developed framework

This section details the applicability and usability experiments of the developed Framework. Firstly, a conceptual verification of its capabilities is presented in the electomobility use case - how the framework can take over orchestrational tasks to provide seamless integration between independent systems with various stakeholders. This demonstration showcases how the extended Local Cloud concept is applicable and how intra- and inter-Cloud orchestration utilized.

Secondly, a technical verification of the framework is also necessary to validate the reference implementation. Scalability and stress testing is necessary to examine the bottlenecks and whether real-time QoS expectations can be facilitated with the current code base. Section 3.2 presents the preliminary results of these tests.

3.1 Verification of the framework principles[1]

The basics of the Local Cloud approach in itself only define network segmentation and locally centralized governance that has restricted applicability in the automation world. However, with the inter-Cloud collaboration architecture and the dynamical orchestration processes introduced by the author, it is now possible to integrate multi-stakeholder scenarios in the Arrowhead framework as well. Here, establishing Local Clouds are tied to business use cases: companies define somewhat closed environments for their operations. In there, the Arrowhead core framework manages systems run-time based on the configuration stemming from operational targets. Currently, these targets are manually created within the core databases, however, future work includes automated choreography generation based on physical processes, see section 4.2.

However, these isolated Local Clouds cannot fulfill new business trends that aim to create an overarching cooperation between enterprise systems (e.g. integration along the value chain). Therefore, as the developed framework is now capable of negotiating servicing terms between Clouds - that then are sanctioned by both governing entities - new opportunities arise.

[1] This subsection is based on the joint demonstration with evopro Inc. detailed in [40].

The developed framework (and its revised fundamentals) was firstly validated in one of todays key development area: the electromobility use case. In this vision, multiple (currently isolated) systems collaborate to achieve smart distribution and management of energy to provide uninterrupted service for the emerging number of electric vehicles. Figure 3.1 details how the various elements of this (tier 2) system of systems are interconnected and can be represented as Arrowhead Local Clouds. In here, a number of stakeholders have to be taken into account: from car manufacturers through charging station infrastructure management up to the connections to the virtual markets of energy. The depicted systems all implement their own sets of application protocols, therefore the whole set of capabilities listed in section 2.1 (Requirements) are necessary to govern this scenario - and even more.

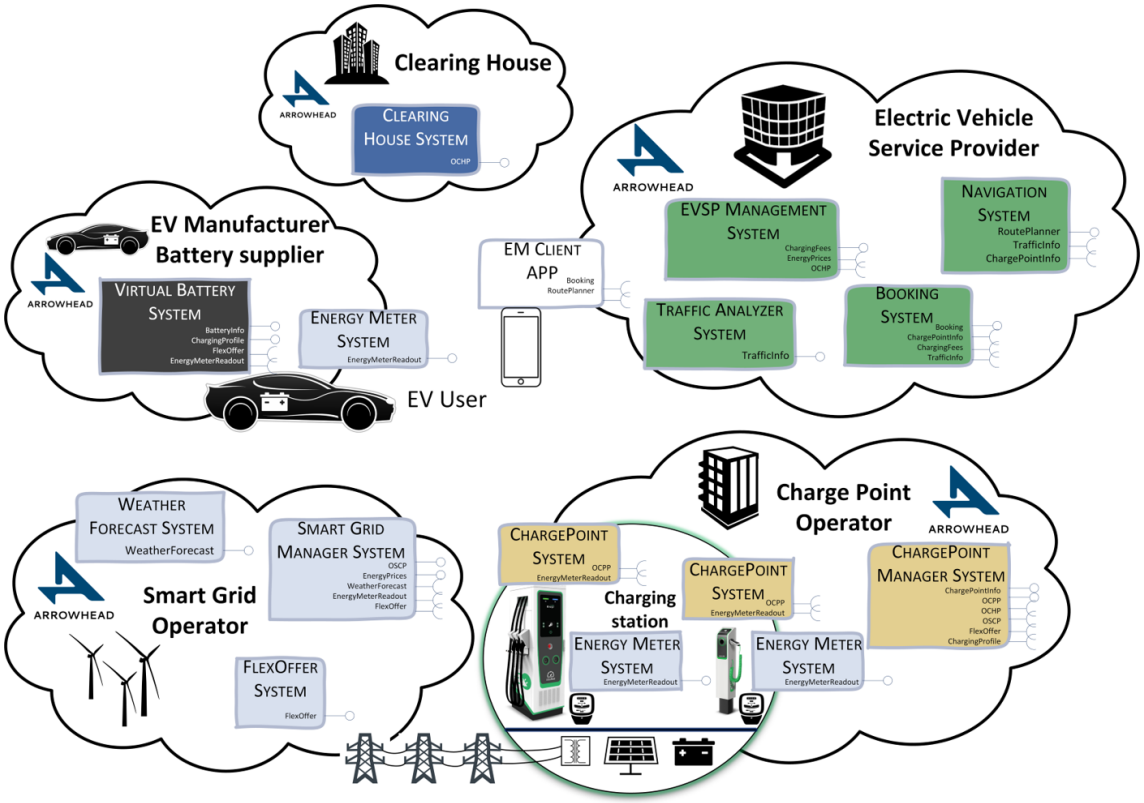

Figure 3.1: *Local Clouds deployed within the electromobility use case*

During this demonstration, the developed framework was used to provide both intra- and inter-Cloud orchestration. The basic scenario is a supposedly simple station reservation case. However, in a scenario where everything is interconnected, a complex process is required to facilitate this simple end user request. This includes fetching the current status of the user's car battery (from its manufacturer) to determine charging parameters and then reserving energy on the grid based on that. Moreover, the choice and reservation of a suitable (possibly near) station is also required - and that might be operated by a different vendor as well. In here, the framework provided the following orchestrational tasks to facilitate all this:

- Default configuration: electric charging stations seeking their currently active management servers after wake-up. This information cannot be hardwired into the stations, since the infrastructure management function might belong to different entities e.g. based on time of

day (e.g. between a municipality and private operators).

- Store-based orchestration: the reservation system fetches the car battery's profile from the manufacturer (based on type metadata) and then reaches out to the smart grid energy market to buy.
- Dynamical orchestration: the infrastructure management system is seeking available (free) charging stations that can facilitate the current reservation request based on various parameters (e.g. car type, geographical location, battery status).

3.2 Usability experiments

3.2.1 Identified test targets and scenarios

Since this framework aims to create run-time governance within automation scenarios, where response time might be limited, it is of high interest to measure the capabilities of the developed architecture since its primary task of orchestration relies on database accesses and internal communications between Core Systems via REST interfaces (to promote distributed deployability).

Therefore, this designed flexibility might have negative effects on system response time that cannot be tolerated in real-time scenarios (e.g. in re-orchestration of failed connections). Currently, there should be three primary platforms for deploying the Core Systems: (i) public servers, (ii) private on-site servers and (iii) embedded controllers (or gateways, e.g. in environmental sensor networks).

Moreover, we have to differentiate between the three types of processes integrated: (i) default configuration-typed (related to Application System startup), (ii) intra-Cloud (and often Store-based) governance and (iii) dynamical (and inter-Cloud) orchestration. Although, these scenarios might be augmented and enhanced later, but the priorities between them should be cleared: the Orchestration Service has to be segmented into at least two categories based on service QoS.

One of them is dedicated towards strict real-time expectations, where orchestration shall not take much time, since physical processes are dependent on it. In here, the telemetry collection, event detection and re-orchestration process of a failure will have an upper limit on execution time. The other scenario, where dynamic matchmaking and lookup is done (even includes inter-Cloud communications), the dynamic aspects are more important than reaction time (e.g. in IoT-like use cases, such as electromobility).

3.2.2 Tests carried out

The current development environment consists of two publicly available servers each one forming a whole Local Cloud. These are deployed on Cent OS running on virtual machines – each limited to 2 GB of RAM and 2 processor cores. The database runs on a MySQL Community server, while the core modules themselves are separate Java executables. The following technologies are

used: (i) Java Jersey [31], (ii) Hibernate Object-Relational Mapper (ORM) [30] and (iii) Grizzly servlet container [32].

To test out the responsiveness of the core framework, I created a simple tool using Java multithreading: it is capable of sending custom amounts and combinations of Service Requests simultaneously and taking note of the response time in a .csv file. This way, the scalability of the implementation can be tested. However, there were restrictions on how many simultaneous connections are allowed to avoid being identified as a Denial of Service attack: the two instances were deployed on the Telecommunications and Media Informatics Department's infrastructure. For future tests, the whole deployment will have to be moved to an unprovisioned network segment.

Nevertheless, I have successfully identified certain nice behavior characteristics and also some limitations of the current implementation. The dynamic orchestration is considered in three scenarios. Each scenario was measured multiple times with an increasing number of parallel requests (10 - 20 -30). The first tests showed that sending in the same request is not a valid test case, since after the first request, response time falls back significantly. Therefore, multiple Service Request Forms were registered in the core framework for each choice on orchestration. For each scenario 3 different Consumers are requesting different Services through (i) default configuration, (ii) Store-based or (iii) dynamical intra- and (iv) inter-Cloud orchestration. The test cases included the followings:

- Only intra-Cloud orchestration: default configuration and Store-based (simulating industrial applications).
- Only intra-Cloud, but dynamical orchestration is used (IoT domains).
- Well established Local Clouds: mainly intra-Cloud orchestration, with occasional inter-Cloud collaborations.
- Excessive inter-Cloud communications (not well established Local Cloud boundaries).

As an example of the results, fig. 3.2 depicts the histograms of the response time for individual requests. These describe the distribution of how much time it takes when 15 simultaneous requests are made to the orchestration, but their contents differ: e.g. 10 of them is for intra-Cloud dynamical orchestration and 5 ends up in inter-Cloud orchestration. This scenario is similar to what we can expect in the verification use case of electromobility described in the previous section.

The experiment resulted in the following identified issues:

- The currently activated logging profile (for debugging) is too aggressive, cripples scalability.
- The virtual machine based server solution has a very slow response time.
- The wakeup time of the virtual machine SQL server is higher than acceptable in certain cases (some Java Database Connections - JDBC- might run into timeout).
- The Store-based orchestration accesses the same tables too fast, therefore the amount of possible parallel connections are limited.
- The inter-Cloud orchestration does add significant amount of extra latency to orchestration: therefore timeout sensitive Services cannot be orchestrated this way.

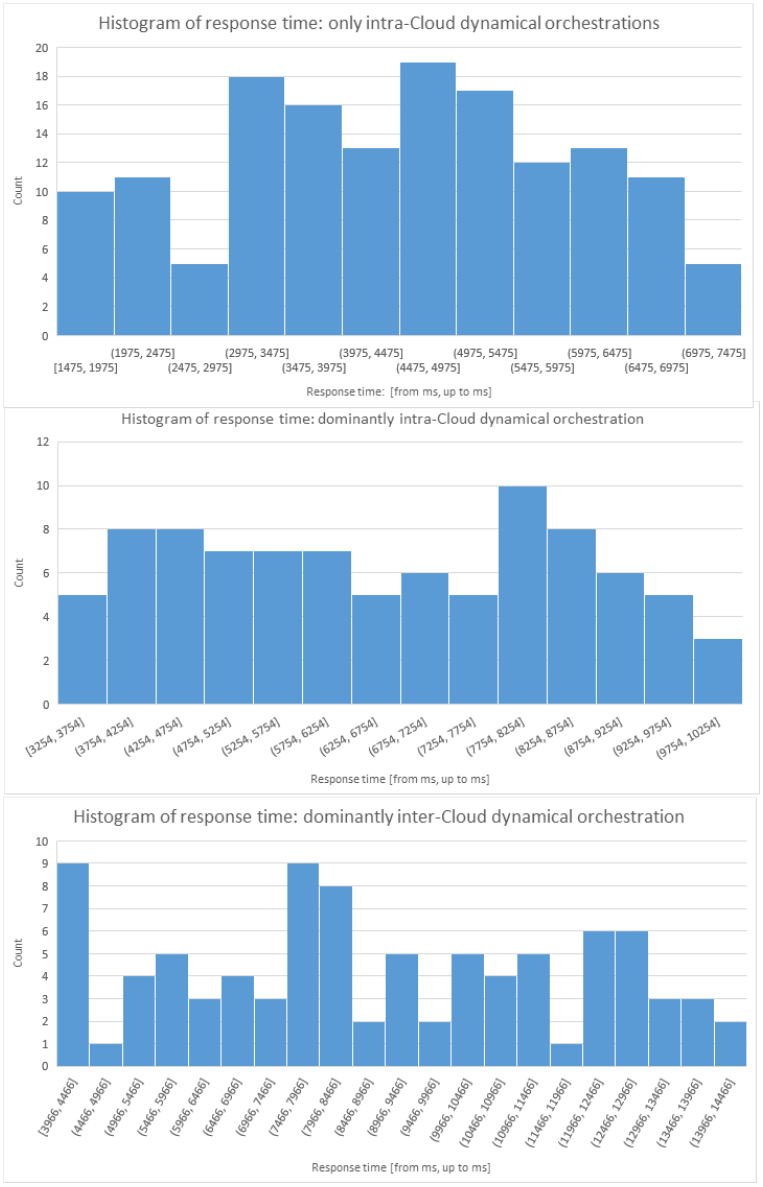

Figure 3.2: *Response time tests of orchestration*

- Various load balancing techniques of the deployment (Grizzly server and VM characteristics) were (re)discovered.
- Currently the Orchestration Store-related tables in the database form the main bottleneck.

Besides these technical issues, the tests concluded that Store-based orchestration has an average of 3.4 seconds when 10 parallel requests are made, and 7. 9 seconds when 30 requests are made. This leads to the conclusion, that this test bed deployment is not yet suitable for real-time management of an industrial SoS as is. However, it is partially acceptable when taking the processes behind the curtains into account. It is also clear, that further work will be required to streamline communications between Core Systems for embedded applications (joining them in one executable), e.g. for the Raspberry Pi target environment.

Chapter 4

Roadmap for future developments

In order to provide an even more complete technical picture than the earlier parts of this document describes, let us go through these enhancements – that are already have their high-level design created, but not yet developed.

Therefore, this section lays out concepts for future work to enhance this framework, since the declared objectives of section 2.1.1 are not completely realized yet. Since this work is designed to support various - yet to be fully described - capabilities and sub-processes, it is of importance to declare design components that will facilitate those. Currently, the framework provides minimalistic capabilities for e.g. resource management, event handling and distillation of SoS choreographies. Also, the data path issues identified for inter-Cloud communications are still yet to be resolved - possibly building on existing solutions.

4.1 Resource management and event handling

As indicated in the previous chapter, real-time Quality of Service is essential within the automation domain. Therefore, resource management has to be done not just between endpoints, but in the Application Systems themselves. Currently, the QoS Manager and QoS Monitor Systems are responsible for that. Reservation is made during the orchestration process, where possible Service Providers are assessed whether they can satisfy the QoS expectations set in the SRF and - after choosing the most appropriate one - the actual reservation is made for the Consumer - Service - Provider trio (as observable on appendix F.0.4). The current implementation of the QoS subsystem is based on custom time triggered Ethernet developed by the Portuguese partners [3]: it uses allocatable time slots using specific messages and traffic shaping on industrial Ethernet.

However, currently the monitoring function is limited to disbanding the current setup, thus proper re-orchestration of a failing servicing instance is yet to be solved. This will require another co-operation between the Core Systems: detecting, identifying and solving events occurring in the Local Cloud. The orchestration process has anchor-points designed to handle such scenarios (by adding new orchestration flags), however the Core System roles and responsibilities in this

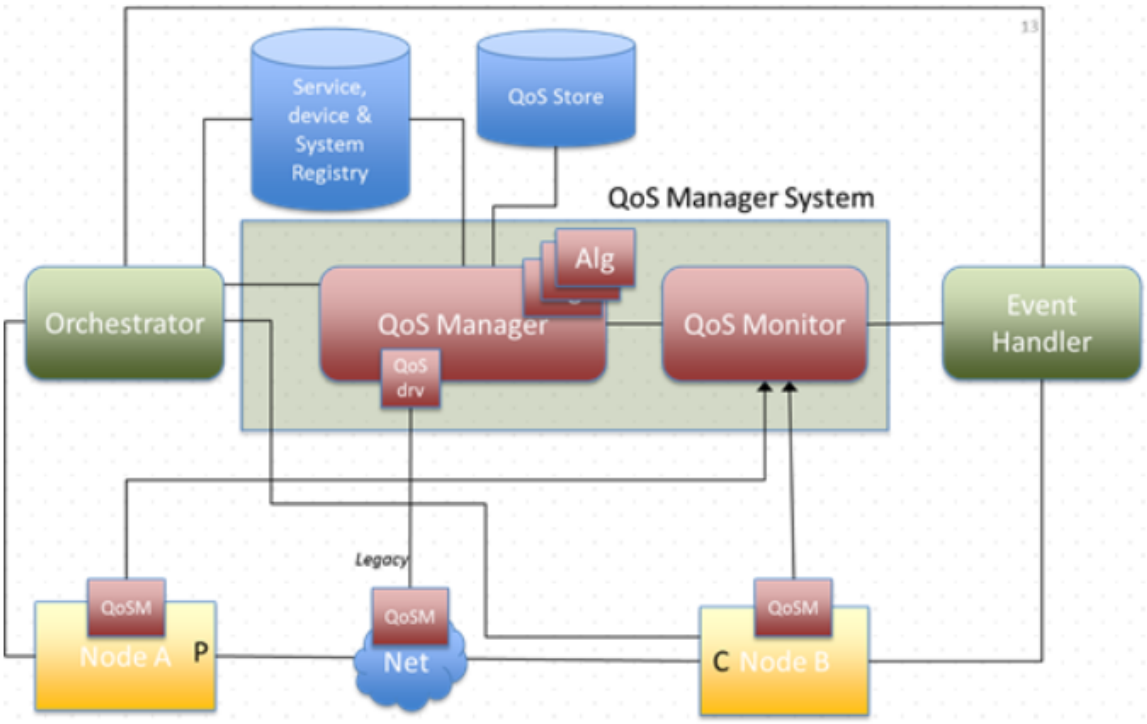

Figure 4.1: *Core Systems interacting for resource management*

matter (between the QoS Monitor, Event Handler System [1] and the Orchestrator) are yet to be cleared. Further issues are to be solved in relation to error handling during translation [17] between Systems or Local Clouds.

Currently, the Orchestrator is request-triggered (not a traditional centralized choreographer). If a servicing instance breaks up (e.g. the Provider vanishes in an instant), it should be the partners' task to signal the appropriate Core Systems (e.g. QoS Monitor) and request re-orchestration. Through this approach, the scalability of a Local Cloud would be maintained, since there should be no centralized monitoring of every Application System in run-time. That would require periodic control traffic between the monitor and all Systems. Nevertheless, this approach puts more responsibility on Application Systems, that could still be an issue (noticing that the servicing partner is not adequate anymore).

4.2 Engineering System-of-Systems[1]

The Core Systems currently run on pre-defined rule-sets (in the Orchestration and Authorization stores and in the Service Registry). They also assume that Application Systems themselves are configured to fulfill their tasks - in somewhat autonomous way. They can fetch their run-time configuration from the Configuration Store - if available. Currently, orchestration refers

[1] This section is based on my article regarding SoS configuration [6].

to a supportive task, namely providing instructions to Application Systems on their service consumption behavior.

What is then more intriguing, how we provide the overall system-of-systems a choreography. This is currently limited to manual planning and configuration of the core databases. However, there should be automated engineering tools within Arrowhead that distillate (e.g. orchestration) rules from operational targets. This is to be tasked with an Plant Configurator whose input is an operational target (e.g. a new manufacturing process) described using appropriate industrial tools using the most suitable view on the devices. Its output then should be placed in the various core databases, as fig. 4.2 suggests. This figure also describes the which core elements are to be affected by the Configurator's presence.

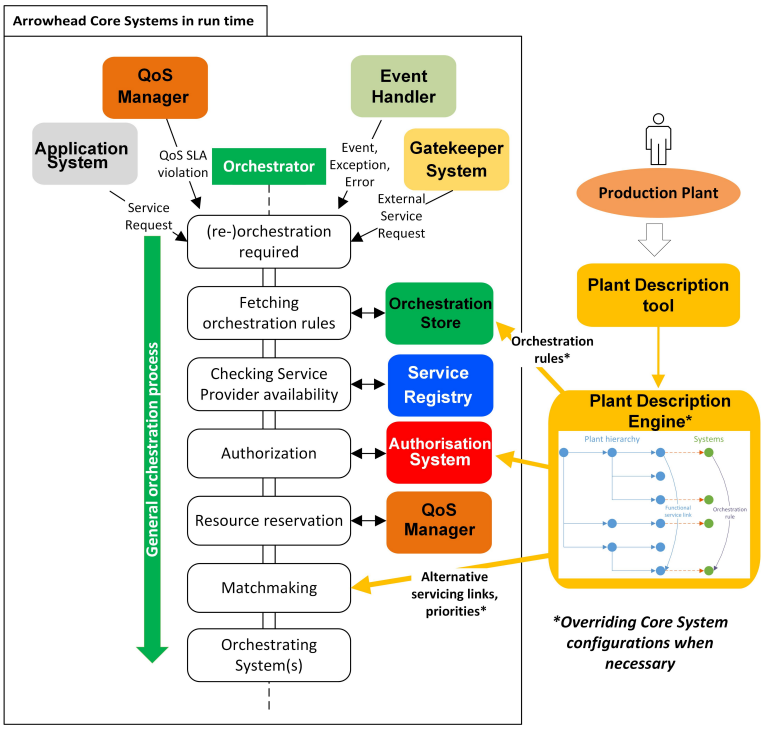

Figure 4.2: *System-of-Systems configurator*

This approach consistent with the choreography solutions of Web Services. The overall configuration of the System-of-System should be optimized for the global target of the given local cloud (e.g. setting a production target for production lines, or optimal energy consumption for a smart building). A such choreographer engine would naturally have to utilize Plant Description related data and its operation must rely on the functional, (physical) architectural and process-oriented mappings of the SoS. This means that the Plant Description in this case will be used to store required, desired or recommended connections between design objects as well (taking over certain relations between Systems from the Orchestration Store).

The decisions and acts of such an engine should be asynchronous to the general operations within a Local Cloud. It should only be invoked to change the general flow in order to enforce new operational targets on the SoS. However, the main difference here compared to Web Services is that not Application Systems will receive the individual choreographies they are required to

follow, rather the Core Systems are re-configured appropriately. After that, there has to be an event broadcast, that every Application System has to request new orchestration (and maybe System configuration).

4.3 Gateways between Local Clouds

Local Clouds are defined by having their own set of various boundaries. These might include having a closed communication network, where addresses are local and firewalls (or even full physical segregation) separates Systems from the outside world. However, within these Local Clouds network visibility and addressability problems are were not considered previously - since it was unnecessary. Yet, with the introduction of inter-Cloud communicational requirements, it is now expected that Application Systems should be able to contact, send to and receive data from external entities (residing in other Local Clouds). It is also required that these interactions happen in a controlled way and should not compromise the Local Cloud's integrity, security or safety.

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

In the Inter-Cloud Negotiations process[2], the last step is building up a data path between the to-be Consumer and Provider that reside in two different Clouds. This data path requires gateways on the edges of both Local Clouds. Such gateways will replicate the servicing partner at the borders and tunnel the communication in-between Clouds, as Fig. 8 suggests.

Configuration of the gateways (e.g. the establishment of Service Provider emulation or creating a data tunnel with the other gateway) has to be part of the process as well as instructing the Service Consumer to connect to its own gateway to start the connection chain. This will be aided by the authorization token concept (with the servicing handshake between Consumer and Provider) that will have to be issued for all three links in a simple scenario. Moreover, as fig. 8 also depicts, there is a distinction between the data (grey) and control (green) planes here and both planes will have to implement a strict protocol (currently called Inter-Cloud Negotiations [ibid]). Meanwhile, the Gatekeeper modules operate on the control plane, Gateways will tunnel communications on the data plane and might will have to be divided into two: external and internal parts.

Between the external- and internal-gateways, a non-routable protocol will have to be implemented. Configuration of the gateways should only be done by their own Orchestrator (through a one-way command port perhaps). This way, the compromise of the external gateway should have no effect on the internals of the Local Clouds. To realize such an advanced tunneling service (that can emulate various Service interfaces, not just REST-based), will require extensive

[2]Described in my Students' Scientific Conference paper of 2015.

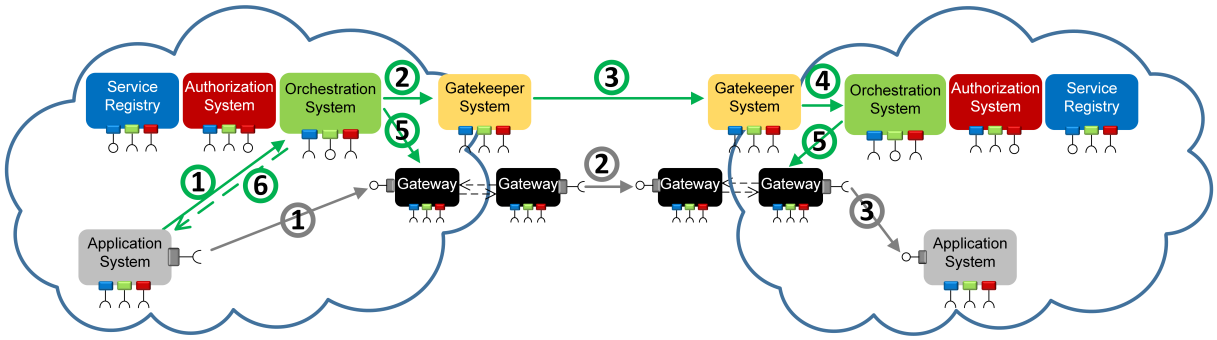

Figure 4.3: *Tunneling connections between Local Clouds*

research and integration of advanced state-of-the-art networking technologies, such as Network Function Virtualization (as discussed in section 1).

Conclusion

To conclude, this paper described the refinement of the Arrowhead framework, my work from the high-level design throughout the implementation up to the preliminary results of the verification process.

Firstly, definitions of the Arrowhead objects and principles were established and polished in order to support various expectations that have arisen through the project. These include Local Clouds, Arrowhead Systems and Services. Common descriptor objects (data structures) were introduced to clarify and formalize communications within the framework.

Secondly, advanced Service Registry and Authorization System capabilities were created to provide the necessary functionality for the revised Orchestrator. This included handling of e.g. Service metadata and a filtering capability within the Service Registry - and an X.509 certificate based solution within the Authorization System (and on Application Systems' side as well). The inter-Cloud architecture was established by the author previously: it relies on the Gatekeeper module.

Thirdly, a centralized governance concept was founded based on various interactions with and between Arrowhead Core Systems. These include - among others - resource allocation tasks, error and event handling, and general management tools for System-of-Systems. The collection and negotiations of the requirements was a major part of this work.

Based on this, and overall high-level design and a roadmap was engineered to cover the whole problem space. As a corner stone of that, the Orchestrator unit was developed to support the run time of Application Systems. Currently, it includes three different processes within its one simple request-response structure, and reliance on the Orchestration Store in some cases.

Finally, the verification of the developed proof-of-concept framework was done on two levels: the overall system design and Local Cloud related principles were showcased and validated in a complex electromobility scenario. It has shown and demonstrated that this version of the framework is capable of facilitating a whole integrated value chain from the end-user (car owners) to the smart grid. Furthermore, preliminary applicability and scalability tests were carried out as well.

Regarding dissemination, this work has resulted in four conference papers and a demonstration at the 43nd International Conference of IEEE Industrial Application Society (2016).

Finally, about current endeavors, the author would like to continue this work, even besides the roadmap for future works described in this paper, as part of his master thesis.

Acknowledgement

I would like to express my deepest gratitude towards my supervisor, Dr. Pál Varga, for the opportunity to work on this project and for his continuous engagement. His guidance and support made it possible to create this work.

I would also like to thank everyone who has worked on this project: the team from the University and from evopro Innovations Inc.

Bibliography

[1] Michele Albano, Luis Lino Ferreira, and Jose Sousa. Event handler system: Publish/subscribe communication for the arrowhead world. In *12th IEEE World Conference on Factory Communication Systems (WFCS)*, 2016.

[2] IPSO Allicance. Ipso smart objects. `http://www.ipso-alliance.org/wp-content/uploads/2016/01/ipso-paper.pdf`, 2016.

[3] Alberto Ballesteros and Julián Proenza. A description of the ftt-se protocol. Technical report, Technical report, DMI, Universitat de les Illes Balears, 2013.

[4] Michael Bell. *Introduction to Service-Oriented Modeling.* Wiley and Sons., 2008.

[5] Fredrik Blomstedt, Luis Lino Ferreira, Markus Klisics, Christos Chrysoulas, Iker Martinez de Soria, Brice Morin, Anatolijs Zabasta, Jens Eliasson, Mats Johansson, and Pal Varga. The arrowhead approach for soa application development and documentation. In *IEEE IECON*, 2014.

[6] Oscar Carlsson, Csaba Hegedűs, Pál Varga, and Jerker Delsing. Organizing iot systems-of-systems from standardized engineering data. In *42nd Annual Conference of IEEE Industrial Electronics Society (IECON)*, Florence, Italy, 2016.

[7] Oscar Carlsson, Pablo Punal Pereira, Jens Eliasson, and Jerker Delsing. Configuration service in cloud based automation systems. In *21th IEEE Conference on Emerging Technologies and Factory Automation*, 2016.

[8] Oscar Carlsson, Daniel Vera, Jerker Delsing, and Bilal Ahmad. Plant descriptions for engineering tool interoperability. In *14th International Conference on Industrial Informatics*, 2016.

[9] Ramon Casellas and Raül Mu noz. Sdn orchestration of openflow and gmpls flexi-grid networks with a stateful hierarchical pce. *J. Opt. Commun. Netw.*, 7:A106–A117, Jan 2015.

[10] Ramon Casellas, Raül Mu noz, Ricardo Martínez, Ricard Vilalta, Lei Liu, Takehiro Tsuritani, Itsuro Morita, Víctor López, Oscar González de Dios, and Juan Pedro Fernández-Palacios. Sdn orchestration of openflow and gmpls flexi-grid networks with a stateful hierarchical pce. *J. Opt. Commun. Netw.*, 7(1):A106–A117, Jan 2015.

[11] S. Cheshire and M. Krochmal. Rfc-6763: Dns-based service discovery, 2013.

[12] Chorevolution. The chorevolution project site. `http://www.chorevolution.eu/bin/view/Main/`, 2016.

[13] The Arrowhead consortia. The arrowhead framework wiki. https://forge.soa4d.org/plugins/mediawiki/wiki/arrowhead-f/index.php/Arrowhead_Framework_Wiki.

[14] MIMOSA consortium. The mimosa project site. http://www.mimosa.org/, 2016.

[15] C. H. Crawford and G. P. Bate. Towards an on demand service-oriented architecture. In . *IBM Systems Journal*, volume 44, pages 81–107, 2005.

[16] Hasan Derhamy, Jens Eliasson, Jerker Delsing, and Peter Priller. A survey of commercial frameworks for the internet of things. In *20th IEEE Conference on Emerging Technologies and Factory Automation*, 2015.

[17] Hasan Derhamy, Pal Varga, Jens Eliasson, Jerker Delsing, and Pablo Punal. Translation error handling for multi-protocol soa systems. In *20th IEEE International Conference on Emerging Technologies and Factory Automation (EFTA)*, 2015.

[18] Remco Dijkman and Marlon Dumas. Service-oriented design: A multi-viewpoint approach. *International journal of cooperative information systems*, 13(04):337–368, 2004.

[19] Scott Dowell, Albert Barreto, James Bret Michael, and Man-Tak Shing. Cloud to cloud interoperability. In *System of Systems Engineering (SoSE), 2011 6th International Conference on*, pages 258–263. IEEE, 2011.

[20] Thomas Erl. *Service-Oriented Architecture (SOA) Concepts, Technology and Design*. 2005.

[21] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. http://www.iotsworldcongress.com/documents/4643185/0/IoT_IBSG_0411FINAL+Cisco.pdf.

[22] Luis Lino Ferreira, Michele Albano, and Jerker Delsing. QoS-as-a-Service in the Local Cloud. In *21th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2016)*, 2016.

[23] Csaba Hegedűs, Dániel Kozma, Gábor Soós, and Pál Varga. Enhancements of the arrowhead framework to refine intcloud service interactions. In *42nd Annual Conference of IEEE Industrial Electronics Society (IECON)*, Florence, Italy, 2016.

[24] Csaba Hegedűs and Pál Varga. Service interaction through gateways for inter-cloud collaboration within the arrowhead framework. In *5th International Conference on Wireless Communications, Vehicular Technology, Information Theory, Aerospace and Electronic Systems (VITAE)*, Hyderabad, India, 2015.

[25] Mario Hermann, Tobias Pentek, and Boris Otto. Design principles for industrie 4.0 scenarios. In *49th International Conference on System Sciences, USA*, 2016.

[26] Yang Hongli, Zhao Xiangpeng, Cai Chao, and Qiu Zongyan. Exploring the connection of choreography and orchestration with exception handling and finalization/compensation. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 81–96. Springer, 2007.

[27] Yang Hongli, Zhao Xiangpeng, Cai Chao, and Qiu Zongyan. Exploring the connection of choreography orchestration with exception handling finalization/compensation. *Formal Techniques for Networked and Distributed Systems - FORTE 2007*, pages 81 – 96, 2007.

[28] Telecommunication Standardization Sector (ITU-T) International Telecommunication Union. X.509: Public-key and attribute certificate frameworks. http://www.itu.int/rec/T-REC-X.509-201210-I/en, 2012.

[29] Michael Jarschel, Thomas Zinner, Tobias Hoßfeld, Phuoc Tran-Gia, and Wolfgang Kellerer. Interfaces, attributes, and use cases: A compass for sdn. *IEEE Communications Magazine*, 52(6):210–217, 2014.

[30] Java. Hibernate orm. http://hibernate.org/orm/.

[31] Java. Jersey: Restful web services in java. https://jersey.java.net/.

[32] Java. Project grizzly. https://grizzly.java.net/.

[33] C. Jennings and Z. Shelby. Media types for sensor markup language (senml) draft-jennings-senml-10. Technical report, IETF, 2013.

[34] Diane Jordan and John Evdemon. Web services business process execution language version 2.0, 2007.

[35] Changbin Liu, Boon Thau Loo, and Yun Mao. Declarative automated cloud resource orchestration. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 26. ACM, 2011.

[36] Pedro Malo, Tiago Teixeira, Bruno Almeida, and Marcio Mateus. Interoperability Repository System for the Internet-of-Things. In *Green Computing and Communications (GreenCom), 2013 IEEE iThings/CPSCom*, 2013.

[37] Raul Muñoz, Ricard Vilalta, Ramon Casellas, Ricardo Martínez, Thomas Szyrkowiec, Achim Autenrieth, Victor López, and Diego López. Sdn/nfv orchestration for dynamic deployment of virtual sdn controllers as vnf for multi-tenant optical networks. In *Optical Fiber Communication Conference*, pages W4J–5. Optical Society of America, 2015.

[38] OASIS. Organization for the advancement of structured information standards. http://www.oasis-open.org.

[39] Intel White Paper. Sdn orchestration layer implementation considerations, 2016.

[40] Bálint Péceli, Csaba Hegedűs, Pál Varga, and Gábor Singler. Integrating an electric vehicle supply equipment with the arrowhead framework. In *42nd Annual Conference of IEEE Industrial Electronics Society (IECON)*, Florence, Italy, 2016.

[41] Sandor Plosz, Csaba Hegedus, and Pal Varga. Advanced Security Considerations in the Arrowhead Framework. In *DECSoS*, pages 1–13, September 2016.

[42] Pablo Punal, Jens Eliasson, and Jerker Delsing. An authentication and access control framework for coap-based internet of things. In *40th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, 2014.

[43] B. Scholten. The road to integration: A guide to applying the isa-95 standard in manufacturing. International Society of Automation, 2007.

[44] Mohammad Hadi Valipour, Bavar Amirzafari, Khashayar Niki Maleki, and Negin Daneshpour. A Brief Survey of Software Architecture Concepts and Service Oriented Architecture. In *2nd IEEE International Conference on Computer Science and Information Technology*, pages 34–38, 2009.

[45] Pal Varga, Fredrik Blomstedt, Luis Lino Ferreira, Jens Eliasson, Mats Johansson, Jerker Delsing, and Iker MartĂnez de Soria. Making system of systems interoperable - the core components of the arrowhead framework. *Journal of Network and Computer Applications*, 2016.

[46] W3C. The world wide web consortium. `https://www.w3.org/`.

[47] W3C. Web services choreography working group. `https://www.w3.org/2002/ws/chor/`, 2009.

[48] A. E. Walsh. *UDDI, SOAP and WSDL: The Web Services Specification Reference Book*. Prentice Hall Professional Technical Reference, 2002.

[49] S. S. Yau and J. Liu. A service matchmaking approach for service-oriented architecture based on service functionalities. `http://dpse.eas.asu.edu/papers/F-Match.pdf`.

[50] Johannes Maria Zaha, Alistair Barrors, Marlon Dumas, and Ter Hofstede. A language for service behavior modeling. *Queensland University of Technology Technical Report*, 2006.

[51] Johannes Maria Zaha, Alistair Barros, Marlon Dumas, and Arthur ter Hofstede. Let's dance: A language for service behavior modeling. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 145–162. Springer, 2006.

[52] Qiu Zongyan, Zhao Xiangpeng, Cai Chao, and Yang Hongli. Towards the theoretical foundation of choreography. 2007.

Appendices

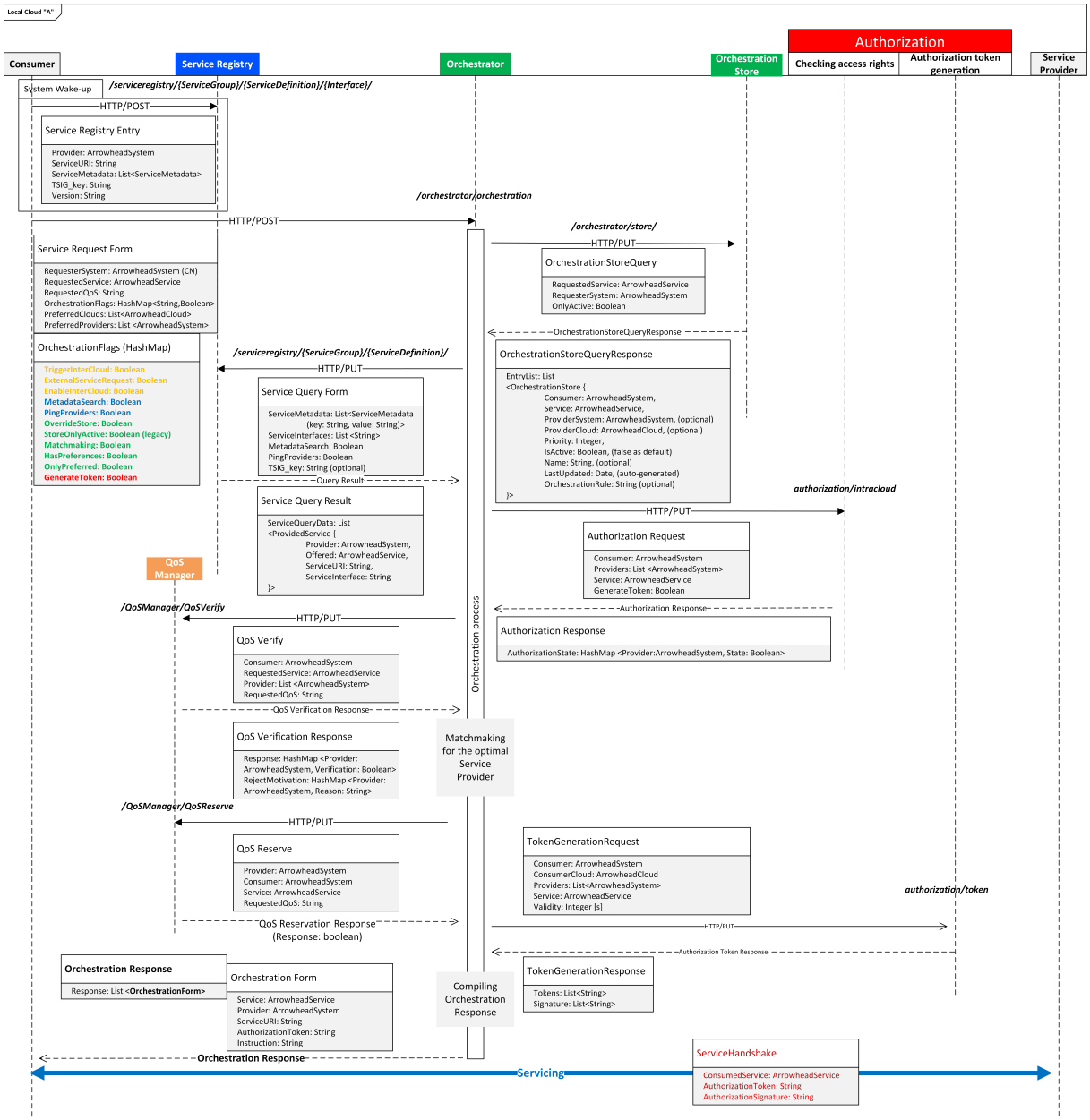

Figure F.0.4: *Dynamical intra-Cloud orchestration message sequence*

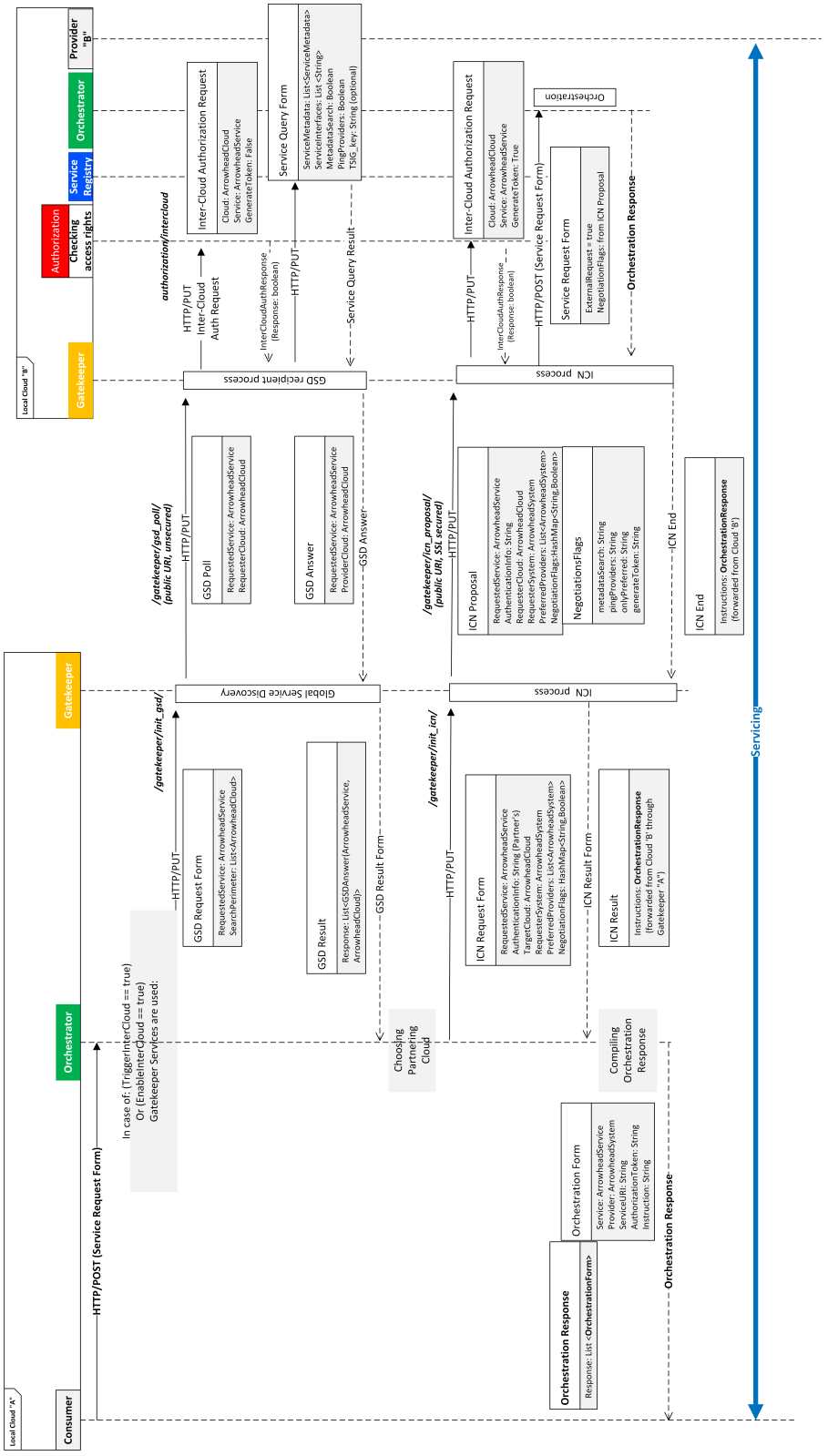

Figure F.0.5: *Inter-Cloud orchestration message sequence*

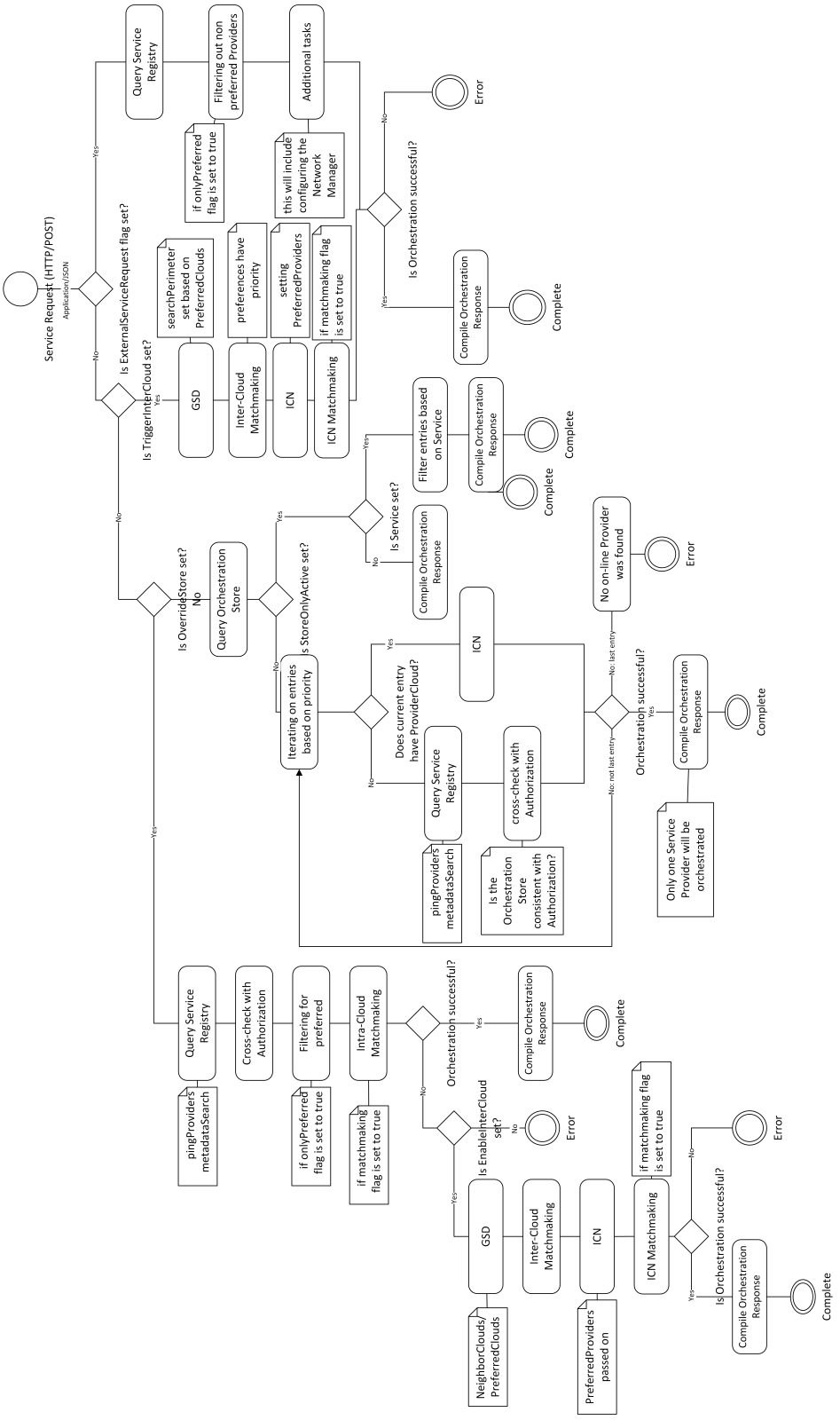

Figure F.0.6: *Inter-Cloud orchestration message sequence*