ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

INTEGRATION OF A SIMULATION PLATFORM FOR ELECTRICAL MOBILITY WITHIN THE ARROWHEAD INTEROPERABILITY FRAMEWORK

Tesi di Laurea in Laboratiorio di Applicazioni Mobili

Supervisor:
Chiar.mo Prof.
LUCIANO BONONI

Co-Supervisors:
Chiar.mo Prof.
TULLIO SALMON CINOTTI
Dott.
MARCO DI FELICE

Presented by:
FEDERICO MONTORI
Matr. 0000668997

Session 3
Academic Year 2013/2014

“Life begins at the end of your comfort zone”

Sommario

Questo documento di tesi riguarda lo sviluppo di un progetto, durante un periodo di più di due anni, portato a termine nell'ambito del Framework Arrowhead e che porta il mio personale contributo in più sezioni. La parte terminale del progetto è stata svolta durante un periodo di visita all'università di Luleå.

Il Progetto Arrowhead è un progetto europeo, appartenente all'associazione ARTEMIS, che si propone di incoraggiare l'utilizzo di nuove tecnologie, quali il fenomeno "Internet of Things", Smart Houses, Electrical Mobility e la produzione di energie rinnovabili, e unificare l'accesso a tutte queste risorse in un unico Framework. Un'applicazione è considerata conforme a tale framework nel momento in cui rispetta il paradigma Service Oriented Architecure ed è in grado di interagire con un preciso insieme di componenti chiamati Arrowhead Core Services.

Il mio personale contributo a tale progetto è dato dallo sviluppo di una serie di API user-friendly, pubblicate sul principale repository del progetto, e dall'integrazione di un sistema legacy all'interno del Framework Arrowhead. L'implementazione di questo sistema legacy è stata iniziata da me nel 2012 e, dopo numerose migliorie portate a termine da un insieme di sviluppatori in UniBO, è stato nuovamente e significativamente modificato quest'anno per poter raggiungere la compatibilità.

Il sistema consiste in una simulazione di uno scenario urbano, ove un certo numero di veicoli elettrici viaggiano lungo le loro rotte, consumano la batteria e, perciò, necessitano di ricaricarsi alle stazioni di ricarica, le quali sono simulate a loro volta. I veicoli elettrici hanno bisogno di utilizzare un meccanismo di prenotazione in modo da potersi ricaricare evitando code dovute alla lunghezza del processo di ricarica.

L'integrazione con il suddetto framework consiste nella pubblicazione dei servizi che il sistema fornisce all'utente finale attraverso l'istanziazione di numerosi Arrowhead Service Provider, in aggiunta a un'applicazione client dimostrativa Arrowhead-compliant in grado di consumare tali servizi.

Abstract

This dissertation document deals with the development of a project, over a span of more than two years, carried out within the scope of the Arrowhead Framework and which bears my personal contribution in several sections. The final part of the project took place during a visiting period at the university of Luleå.

The Arrowhead Project is an European project, belonging to the ARTEMIS association, which aims to foster new technologies and unify the access to them into an unique framework. Such technologies include the Internet of Things phenomenon, Smart Houses, Electrical Mobility and renewable energy production. An application is considered compliant with such framework when it respects the Service Oriented Architecture paradigm and it is able to interact with a set of defined components called Arrowhead Core Services.

My personal contribution to this project is given by the development of several user-friendly API, published in the project's main repository, and the integration of a legacy system within the Arrowhead Framework. The implementation of this legacy system was initiated by me in 2012 and, after many improvements carried out by several developers in UniBO, it has been again significantly modified this year in order to achieve compatibility.

The system consists of a simulation of an urban scenario where a certain amount of electrical vehicles are traveling along their specified routes. The vehicles are consuming their battery and, thus, need to recharge at the charging stations. The electrical vehicles need to use a reservation mechanism to be able to recharge and avoid waiting lines, due to the long recharge process.

The integration with the above mentioned framework consists in the publication of the services that the system provides to the end users through the instantiation of several Arrowhead Service Producers, together with a demo Arrowhead-compliant client application able to consume such services.

Contents

List of Figures

Chapter 1

Introduction

1.1 Service Oriented Architectures

Nowadays, all the human-machine interactions and the machine-machine interactions are based on services. There is an increasing need of informations and those informations must rely on systems capable of providing them anytime. A system providing this kind of informations to any other system or user who may need it, provides indeed a service.

1.1.1 Definitions

In computer science, a service is a self-contained unit of functionality which can be combined with others to provide a wide range of operations constituting the complete set of use cases of an application [10]. Service Oriented Architectures (SOA) are based on this atomic concept.

SOA is a design pattern of common use and independent from the technologies which aims to provide and organize different and distributed capabilities that can be under the control of different domains and vendors. Moreover, it provides an unique approach to discover, offer and interact with these heterogeneous services without any underlying technology constraint [29]. Often it can be seen as a Middleware performing communication mediation between the parts.

Services must be uniquely discoverable and should provide chunks of metadata which specify how to interact with them and how data are represented and distributed. In particular, service metadata describe in sufficient detail in which format the data is provided (generally programmers use to represent data in XML [41] wrapped in exhaustive description containers), how is the service internally defined within the framework (often WSDL [44] is used) and which communication protocols are used and how (an example may be SOAP [45]). Using metadata, application systems aiming to discover and use the service shall be able to configure dynamically to adapt their interface and maintain coherence and integrity. Thus, any service consumer or aggregator would be aware of which services are offered by discovery and can easily interface with them without any knowledge of the service's implementation. The essential concept is the contrast between the complexity of the implementation, which can be of any kind, and its simple interface together with a good interface description [30].

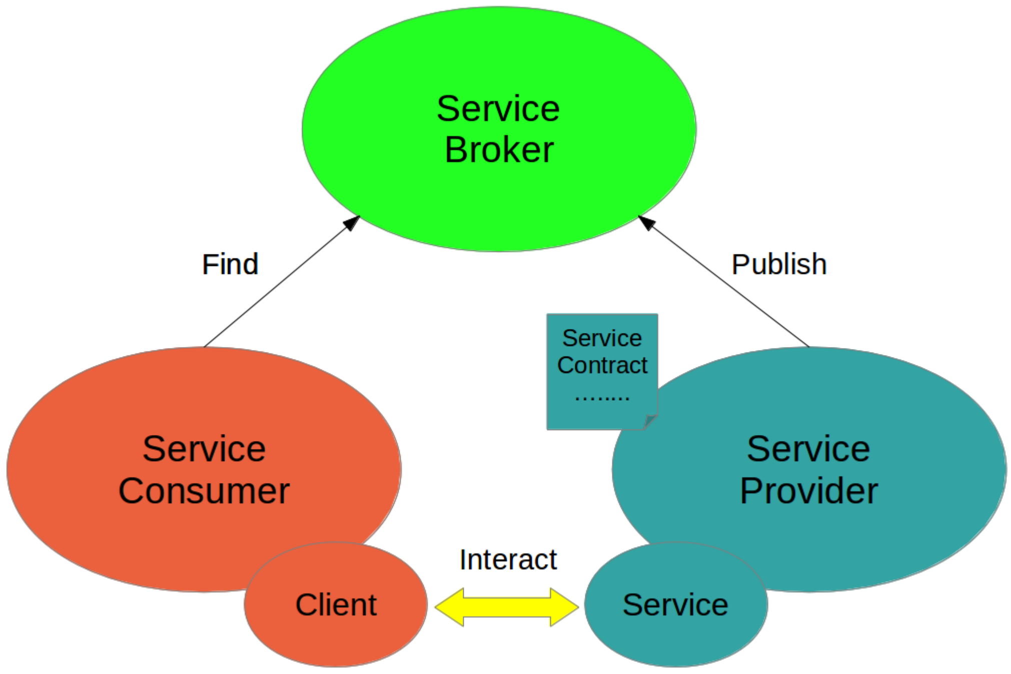


Figure 1.1: The skeleton of a SOA concept: a client discovers and consumes the service offered by a provider [1].

1.1.2 Principles

IBM developers state that the real integration killer is the multiplicity of the interface. Given n different cooperating systems with a different access interface each, we may face a complexity of $n(n-1)$ different implemented interactions. Furthermore, if a new system needs to be integrated, it requires other $2n$ access interfaces to be built from scratch [30]. A SOA framework prevents these problems and provides the application developers with an unique access interface.

The first and foremost concept that a SOA integration must follow is leveraging existing assets. Starting over a new SOA compatible system requires a huge amount of money and resources and, when the system already exists (i.e. a legacy system), it cannot be thrown away but it has to be integrated using a component often defined as adapter. The system, however, is supposed to acquire more and more maturity over time by getting its parts incrementally replaced.

Moreover, there are many principles that a SOA architecture must follow:

- Services shall adhere to a **communication agreement**, specified in the proper documentation language.

- Services have to be **loosely coupled**, so they must not depend on each other and have to be as atomic as possible.

- Services must maintain **abstraction**, so it should not be possible to get their logic from outside. This also helps to accomplish the design principle known as "Service Abstraction Design Principle" which states that services must not develop any kind of mutual dependancy.

- Services must be **fine grained**, to promote **reusability** and **aggregation**.

- Services have to be **autonomous**, so they have the complete control over their logic.

- Services have to as more **stateless** as possible, so they minimize the resource consumption.

- Services have to be **discoverable** from anywhere within the SOA network.

- Services have to be compliant and aware of each other in order to enable their **modularity**.

1.1.3 Design Concepts

A SOA framework has been defined as a structure of five horizontal, abstraction-based layers called Architecture Building Blocks (ABB), here listed starting from the less abstract [31]:

1. **Operational System Layer**: the actual runtime environment where all the back-end applications and the systems reside and run.

2. **Component Layer**: software components, libraries and tools that need to be built to support the realization and implementation of the services.

3. **Services Layer**: single services needed to run the main application.

4. **Business Process Layer**: services aggregated to give shape to the main functionalities required from the application.

5. **Consumer Layer**: Access for the end user to the functionalities provided by the application which may consist of a GUI.

Furthermore, there are four transversal layers that should be present at all the implementation levels in order to grant the efficiency of the whole system:

- **Integration Layer**: the integration at every layer should be granted among the parts, otherwise the interoperability will not be possible.
- **Quality of Service**: it is constituted by security, availability, configuration, performance, monitoring, management and many other capabilities which are part of the Non-Functional Requirements.
- **Information**: it is the layer responsible of collecting all the business information.
- **Governance**: it is a central point where all the documents regarding policies and agreements are stored.

It is to be highlighted that a SOA, as it operates with a lot of different applications from disparate domains, does not provide an interface to its services in terms of API, but rather in terms of protocols and functionalities. An entry point to one of these services is called *endpoint*.

Web services

There is a wide variety of Web services running SOA nowadays. Web services are actually the most common and the easiest platforms where to make services available. Commonly, it is possible to enable such services to be SOA-compliant using some sort of wrappers or adapters. In particular, actors using Web services in SOA can play one or both of the following roles (in subsection 3.2 those concept will be defined in detail in the scope of the present project's domain)[15]:

- **Service Provider**: it creates a Web service and publishes it to the service registry together with all the consistent metadata. It is its responsibility to decide who has the right to access it, for which price to sell it, if present, what category to assign to it, which brokers are enabled to be mediators.
- **Service Consumer**: it is part of an application that makes use of the services. It is its responsibility to look up the broker's service registry for the desired service or class of services and perform the actual consumption when binded to it.

1.1.4 Technologies

SOA paradigm has been instituted by and introduced several various technologies and protocols that enabled fast and organized development of compliant applications. Frequently, those technologies have become a standard in various SOA domains. Some of the most important are listed below:

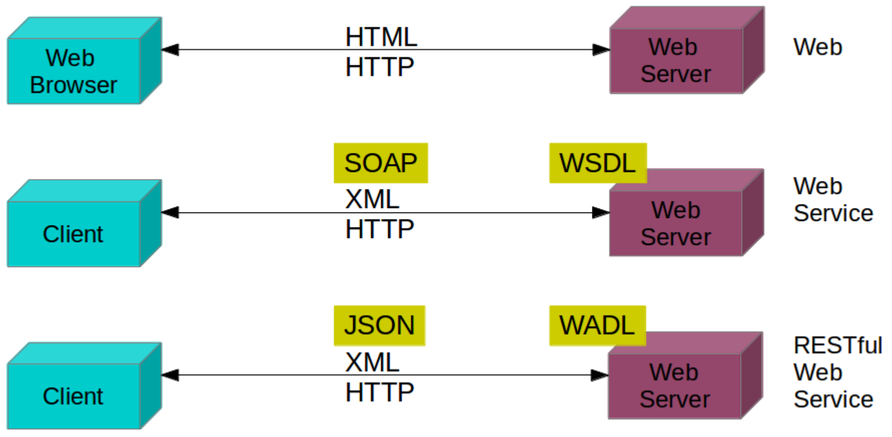


Figure 1.2: An example of how protocols and technologies can combine to create different services, from a simple web access to a complete RESTful Webservice [1].

- **CoAP**[46]: it is a specialized web transfer protocol (application level) for constrained nodes and constrained networks in the Internet of Things. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation and is intended to be used by small and simple entities such as wireless sensor network nodes.
- **OPC-UA**[47]: it is a platform independent service-oriented architecture that integrates all the functionality of the individual OPC Classic specifications into one extensible framework.
- **MQTT**[48]: it is a M2M and IoT connectivity protocol (application level). It was designed as an extremely lightweight publish/subscribe messaging transport and now it is an OASIS standard.
- **XMPP**[49]: it is a XML-based communication protocol for message-based middleware.
- **DDS**[50]: it is a M2M middleware standard designed to enable scalable, real-time and high performance data exchange between embedded devices such as mobile operating systems, vehicles built-in software and health care dependant software.

- **SOAP**: it is a protocol specification and an architectural style, directly derived for the Remote Procedure Call (RPC) paradigm, that aims to define how data are exchanged in Web services networks. It is relying on XML for the representation of data and uses application level protocols, such as HTTP and SMTP, for the data transmission. The XML structure is contained in a node called *envelope*, which may be divided for example in *header* and *body*. Furthermore, a set of encoding rules for the application-specific data, requests and responses has to be defined. Once these elements are known among the parts, SOAP is independent from the programming model and can run over almost every transport layer (or higher) network protocol. Unfortunately, as XML is considered to be verbose, applications that need good performances usually are leaded to use REST.

- **REST**: it is, just as SOAP, a protocol specification and an architectural style that aims to standardize the exchange of data in Web services by defining a certain number of operations. It uses HTTP as transfer protocol and specifies the action that a client can perform in four operations: GET (to get the list of a collection or an element's representation), PUT (to replace an element with another given in input), POST (to add an element given in input) and DELETE (to erase the selected value). In general, nowadays the wide majority of the RESTful applications use to perform a POST in order to perform a GET, a PUT or a DELETE, as they differ just for the server implementation, which is always decided by the developer, though this behaviour is out of the protocol. In order to be RESTful, an application must satisfy the following constraints: the model should be client-server, the server must be stateless (though it can still store informations in external structures such as databases), the responses must define themselves as cacheable or not in order to prevent the client from caching a resource that can change over time, messages should be self-descriptive, the system must be layered and the resources must not be tied to a specific representation.

- **CORBA**[51]: it is, like SOAP, an architectural style derived from RPC which adds Object Oriented features to the information exchange. It aims to enable the information transmission by accessing to remote objects as if they were local. This communication is held per object by an Object Request Broker (ORB) which knows the structure of the request to be performed thanks to the interface, coded in an Interface Description Language (IDL) and provided by the server.

It is important to distinguish between a Service Object-Oriented Architecture (SOOA) and a Service Protocol-Oriented Architecture (SPOA)[16]. According to the first one, services are following a call/response paradigm and are remote objects that are accepting remote invocations and are documented using a codebase annotation. SOOA is protocol neutral and does not require the requester to bind to a certain proxy, because it already holds the object to be filled. According to the second one, services are following a read/write paradigm and are described using some passive service description such as WSDL documents for Web services or IDL documents for CORBA. With SPOA, the requester needs to generate a proxy and bind it to the provider, forwarding to it all its calls.

1.2 The Arrowhead Project

The European Project Arrowhead [33], developed within the ARTEMIS association [34], aims to maximize the efficiency and the flexibility of the usage of energy by leveraging the availability and the interoperability of smart energy resources. It pushes cooperative automation through the centralization and the standardization of these services using SOA core functionalities.

1.2.1 Story and Relevance of the Call

The project started in May 1st 2013 and has a duration of four years. It find its aim on the fact that the society is facing both energy and competitiveness challenges.

These challenges can be found in multiple sides: the production side, which is affected by raw material changes, to which it must adapt, and by environmental impact, the city side, in which energy consumption must be reduced, and the energy itself side, for which enabling renewable production, grid safety and reduced emissions has been a common issue over the years. The key concept is interaction and interoperability, as all energy producers, energy consumers, involved system, domains and so on has to be dynamically interoperable to cooperate. Thus, a global system for communication and aggregation should be enabled, indeed the state of the art suggests an orientation towards Service Oriented Architectures and Internet of Things. The first one, as stated in section 1.1, grants an unique interface and the automatic interaction among systems, while the second one, as stated in section 1.1 of my previous work [17], is the key for the dialogue between the physical world and the information world.

The concept of Smart Environments [6] takes place in a "context", which is defined [4] as any information that can be used to characterize any relevant entity. An entity, in those cases, can be any object or subject capable of holding relevant information: a person, a system, a sensor and so on so forth. The Smart Environment itself constitutes a physical world which interweaves over time all its entities using sensors, actuators, displays and other embedded computational elements and transforms their characteristics and their states in useful informations reachable by any user of that same system [3].

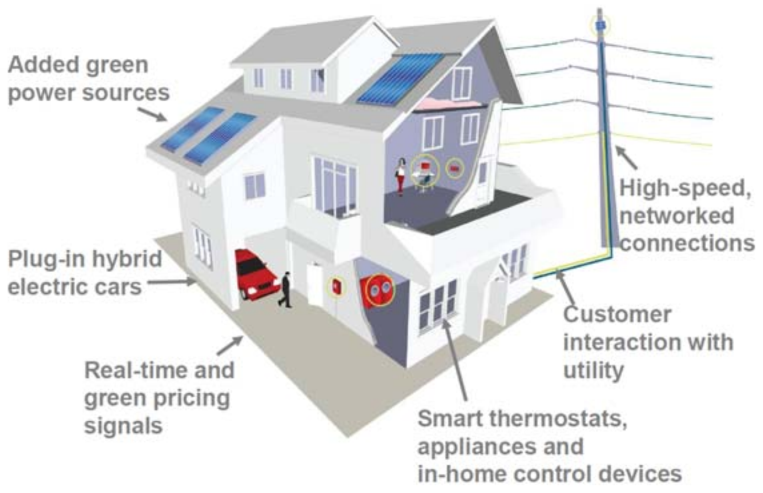

Added green power sources
High-speed, networked connections
Plug-in hybrid electric cars
Customer interaction with utility
Real-time and green pricing signals
Smart thermostats, appliances and in-home control devices

Figure 1.3: An example of Smart Environment: a Smart House

Beyond these theoretical concept, the aim of Arrowhead is [25]:

- Provide a technical framework adapted in terms of functions and performances.

- Propose solutions for migration.

- Implement and evaluate the cooperative automation through real experimentations in applicative domains: electro-mobility, smart buildings, infrastructures and smart cities, industrial production, energy production and energy virtual market.

- Point out the accessible innovations thanks to new services.

- Lead the way to further standardization work.

To reach these goals, all the Arrowhead partners put effort in the usage of the above mentioned framework following the SOA paradigm, that is to say, the exchange of service among loosely coupled actors in order to render as global as possible the so-enabled network. This will bring certainly a numerous variety of benefits, for instance the increase of flexibility, automation, man-machine interaction and adaptation to marked demands.

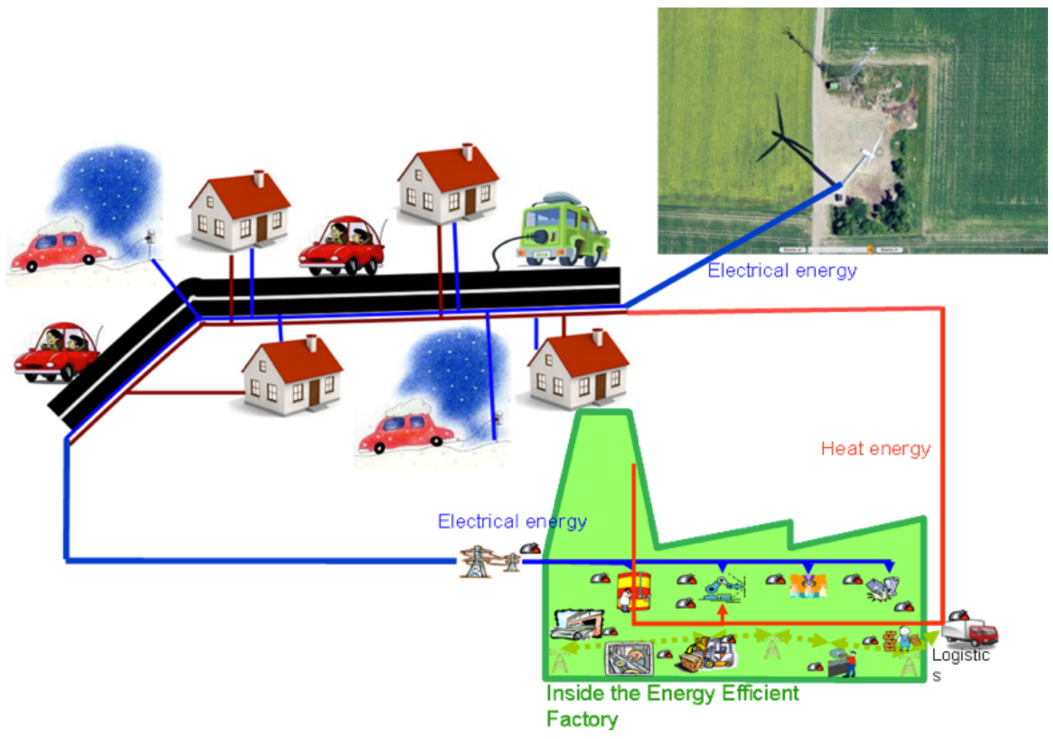


Figure 1.4: Scheme of the improvements that the Arrowhead project is aiming to bring through collaborative automation innovations [25].

1.2.2 Fundamental Concepts

The increasing involvement of billions of new devices producing and consuming informations in various automation tasks is requiring more and more interoperation. The flexibility that the Arrowhead framework aims to address makes this interoperation not only easier, but possible even in the future, where the number of entities involved is predicted to grow dramatically.

In particular, the Arrowhead project takes place in five energy related application domains:

1. Production (manufacturing and process).

2. Smart buildings and infrastructures.

3. Electro-Mobility (or E-Mobility).

4. Energy production.

5. Virtual market of energy.

For those application domains, Arrowhead designs a collaborative automation which is intra-domain at a first level, and cross-domain at a further level.

Therefore, we can state that the Arrowhead vision is, in general to enable collaborative automation by networked embedded devices and thus their interoperability and integrability, inasmuch as a service oriented infrastructure must assume those concepts as basis.

In a more practical vision, the systems running on the above mentioned embedded network devices should interface with three common entities (sometimes aggregated in an unique compound element) using a standard designed framework which can reduce the developer's effort by the 75%. These common parts are referred to, in the theory of service base frameworks, as:

- Information Infrastructure (II), the infrastructure capable of knowledge about which services exist.
- System Management (SM), the infrastructure capable of knowledge about which services are (or should be) connected to whom.
- Information Assurance (IA), the infrastructure capable of knowledge about which services are allowed to exist and who is allowed to consume them.

The figure 1.5 shows at a glance how the systems are expected to be integrated in the Arrowhead service framework.

The current state of the art for production and energy automation is based on the ISA'95 standard paradigm, which integrates service oriented technologies in some parts and mainly inter-domain, especially through the support of OPC-UA from major vendors. This is not true regarding smart buildings and E-Mobility, due to the heterogeneity of vendors and standards. There have been few common standards which are emerging proposed by few past projects such as "Internet of Energy" [32], founded on the Internet of Things concept and mainly addressed for the integration of the smart grid, the electrical grid capable of energy distribution to every source and every destination due to the amount required. The addressing of a common standard, also cross-domain, would reach also one of the ARTEMIS' main targets and a crucial point where the technical barriers are removed. Thus, a set of test beds and tools are also required to drive the application developers (the Pilot Work Packages in the scope of Arrowhead) in a better understanding and an easy integration.

The Arrowhead priority resides in the identification of a number of gaps in the common state of the art and address the Pilot Work Packages to fill them. The common gaps and solutions are listed below:

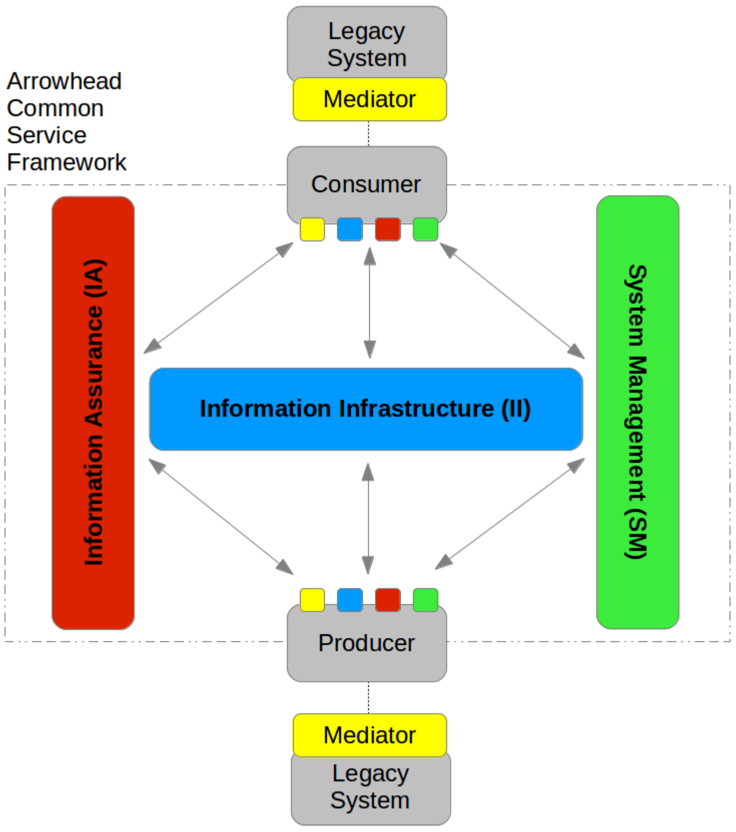

Legacy System
Mediator
Arrowhead Common Service Framework
Consumer
Information Assurance (IA)
Information Infrastructure (II)
System Management (SM)
Producer
Mediator
Legacy System

Figure 1.5: Scheme representing how services should interact in the Arrowhead framework.

- Energy management for self sufficient smart devices, achieved by the development of local monitoring device energy management systems.
- Communication from enclosed location, achieved by enabling transmission of data to the outer world through industrial encapsulated systems.
- Methodology tools and technology for a cost effective development of the core systems.
- Local communication and identification technology.

Identifying the gaps in each technology is essential to enable the natural flow from an experimental and research technology to the market.

Each Pilot Work Package will develop a significant demonstration showing the filling of that gap which is organized through a chain of tasks, each assigned to

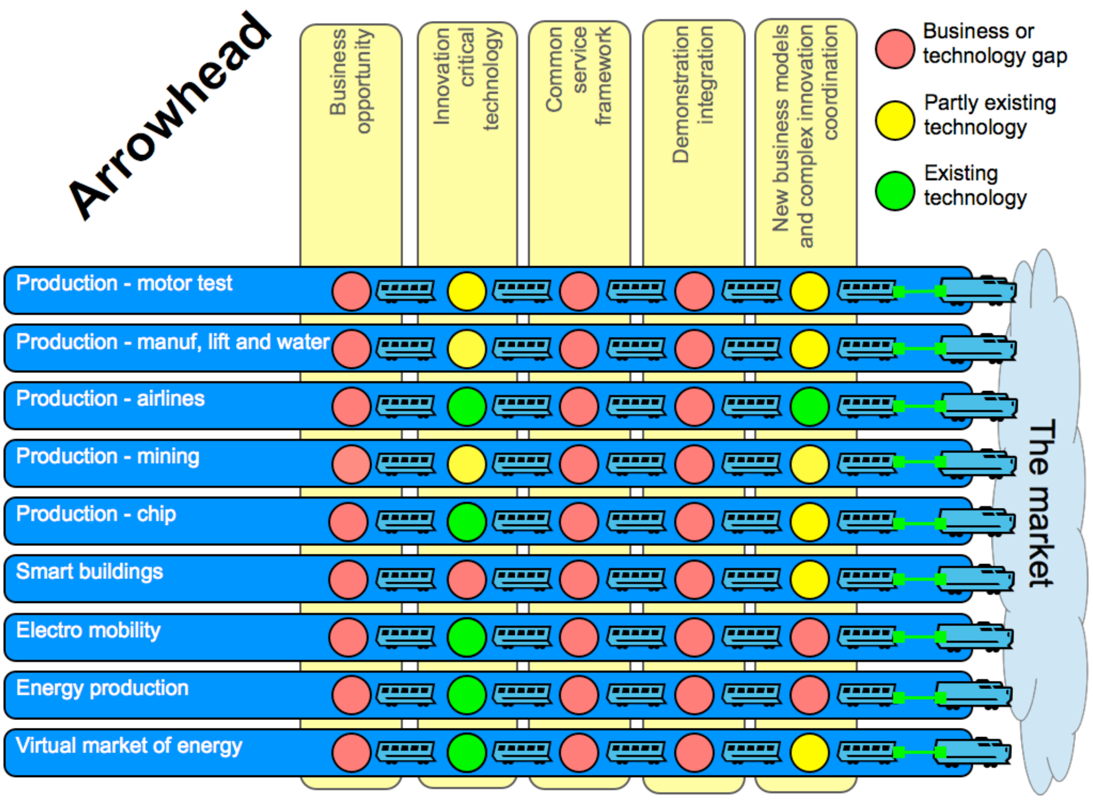


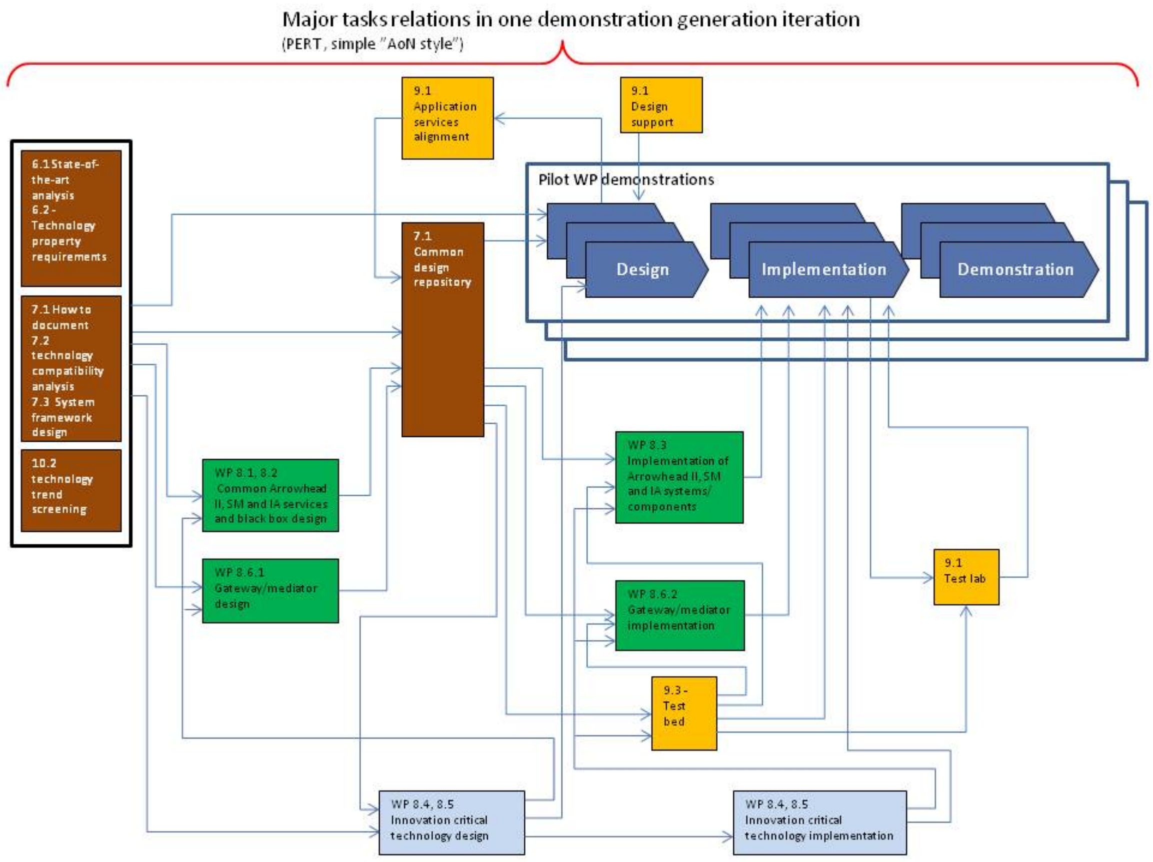
Figure 1.6: Arrowhead strategy against the state of the art [25].

one or more Arrowhead partner. The methodology heart of the demonstration is a production articulated in three phases, the output of each will be the input of the technology R&D, which subsequently feeds the pilot for the creation of the next generation demonstration. Furthermore, the Pilot Work Packages are accompanied by four Technology Work Packages, which are designed to capture the domain specific and the general application requirements to the core services. Due to the size of the Arrowhead project itself, it is expected that a very structured and intensive dissemination of both technologies and demonstrations will lead to a market and society trust. A special Work Package has been instituted for these purposes.

1.2.3 Work Plan

Based on the project organization, as stated before, the effort of providing demonstrations belonging to different working domains is assigned to five pilot Work Packages (WP 1-5). These ones are expected to process the customers' needs and the gaps existing in the current state of the art concerning their proper domains

and act as vehicle of technological progress for the overall project. The following four Work Packages (WP 6-9) are devoted to the necessary common technology development, support, documentation and analysis. Furthermore, the two following Work Packages (WP 10-11) are designed to foster innovation based on the Arrowhead technologies in the energy domain. Finally, there is one Work Package (WP 12) which is devoted to the project management and organizes the components to produce their demonstration in three different iterations, first at a simulation level, then at a domain-specific interaction level, and at last at a cross-domain level.
Figure 1.7 shows how all the WPs should interact to produce their output in the scope of their different tasks.



Major tasks relations in one demonstration generation iteration
(PERT, simple "AoN style")
9.1 Application services alignment
9.1 Design support
Pilot WP demonstrations
Design
Implementation
Demonstration
6.1 State-of-the-art analysis 6.2 - Technology property requirements
7.1 How to document 7.2 technology compatibility analysis 7.3 System framework design
10.2 technology trend screening
7.1 Common design repository
WP 8.1, 8.2 Common Arrowhead II, SM and IA services and black box design
WP 8.6.1 Gateway/mediator design
WP 8.3 Implementation of Arrowhead II, SM and IA systems/ components
WP 8.6.2 Gateway/mediator implementation
9.1 Test lab
9.3 - Test bed
WP 8.4, 8.5 Innovation critical technology design
WP 8.4, 8.5 Innovation critical technology implementation

Figure 1.7: Arrowhead strategy against the state of the art [25].

WP 1 - Pilot Domain: Production (Process and Manufacturing)

The production pilot must prove the efficiency improvements enabled by collaborative automation and service oriented frameworks. This spans over a wide number of application areas concerning enterprises and granularities: from the sensor itself to the enterprise application. It analyzes new business cases concerning interoperability, what are the impacts on people and what are the opportunities in production and in processes. The WP leader is SE. Tasks:

- Task 1.1 - Engines Business
- Task 1.2 - Manufacturing of Electrical Enclosures
- Task 1.3 - Lift Machine Efficiency
- Task 1.4 - Water Distribution
- Task 1.5 - Aircraft Maintenance
- Task 1.6 - Mining Condition Monitoring
- Task 1.7 - Collaborative Engineering for Assembly Automation
- Task 1.8 - Self Condition Monitoring Mobile Machinery
- Task 1.9 - 3D Localization
- Task 1.10 - Condition Monitoring of Transportation Systems
- Task 1.11 - Manufacturing in the Cloud

WP 2 - Pilot Domain: Smart Buildings and Infrastructure

Using the technologies provided mainly by IoT, the infrastructure pilot must foster the usage of smart sensors and actuators that can make users and customers able to interact simply. This happens at different granularities: Smart Houses, Smart Districts, Smart Cities; all of them can make the user aware of energy consumption, management and others. On one hand, the WP operates in the improvement of public services, on the other hand it develops solutions for the deployment of networked embedded systems in a domestic context. The WP leader is Acciona. Tasks:

- Task 2.1 - Energy Efficiency in Buildings
- Task 2.2 - Eco-sufficient Home
- Task 2.3 - Intelligent Urban Lightning
- Task 2.4 - Virtual Control Rooms for Energy Efficiency

WP 3 - Pilot Domain: Electro Mobility

There has been many issues already addressed in past projects, such as IoE, regarding the charging infrastructure of electrical vehicles. Many of these problems were about the energy balance and the vehicle-to-grid operations as well as the lack of charging infrastructures in rural areas. Unfortunately, there has been a lack of demonstrations and still users are unaware of the energy market, which provides low energy costs during the off-peak hours. The mobility pilot will focus on the development of innovative services towards the smart grid and must be able to provide physical demonstrations. The WP leader is CRF. Tasks:

- Task 3.1 - Slow Recharge Stations in Private Environments
- Task 3.2 - Longer Terms Technologies for the Recharging Infrastructures
- Task 3.3 - Device to Cloud Mapping for Electric Mobility

WP 4 - Pilot Domain: Energy Production and End-User Services

The energy pilot aims to open the real time information flow between the energy producer and the energy consumer, which is the end user. The goal is mainly constituted by enabling a distribution network that works over cities and makes possible the passage between a research topic and a real market innovation. The WP leader is Abelko. Tasks:

- Task 4.1 - End User Service - Macro and Micro Perspective
- Task 4.2 - Optimization of Co-Regeneration Systems

WP 5 - Pilot Domain: Virtual Market of Energy

The aim of the market pilot takes place in the TotalFlex [38] energy market, which offers flexible offers to the customers depending on the hour and other various parameters through a service called FlexOffer. In particular, it has to be determined if this paradigm fits with the Arrowhead concepts, a common service interface for different domains, scalability of the infrastructure, aggregation for multiple domains. The WP leader is AAU. Tasks:

- Task 5.1 - Architectural Design and Implementation of Interfaces
- Task 5.2 - Demonstrators of the Virtual Energy Market
- Task 5.3 - Integrated Energy Market

WP 6 - Technology Analysis, State of the Art and Requirements

The aim of this WP is to identify common technology base and interoperability between applications. It is intended to look for, as much as possible, technologies within former international and national funded projects. The WP leader is VTT. Tasks:

- Task 6.1 - State of the Art
- Task 6.2 - Technology Property Requirements for Market Innovation

WP 7 - Interoperability and Integrability Framework

The aim of the WP is to provide an unique way to document every system within the project, document the generic design guidelines and design patterns used and produce templates which other WPs shall follow. Basically, it addresses which features and which functions really need to be common in order to make two system really interoperable. The WP leader is AITIA. Tasks:

- Task 7.1 - Technical Framework Design Pattern
- Task 7.2 - Basic Technology Compatibility Analysis
- Task 7.3 - System Framework Design
- Task 7.4 - Engineering and Operation Methodology

WP 8 - Interoperability and Integrability Service Specification and Common Components Design and Implementation

This WP takes its inputs from the two previous WPs: it chooses a set of different solutions for the low level communication protocols, especially through components of the core systems, minimizing when possible the actual number of solutions and developing interoperability between them. The WP Leader is CEA. Tasks:

- Task 8.1 - Common Arrowhead II, SM and IA Services
- Task 8.2 - Arrowhead II, SM and IA Black Box Design
- Task 8.3 - Implementation of Arrowhead II, SM and IA systems/components
- Task 8.4 - Innovation Critical Pilot Specific Communication Technology
- Task 8.5 - Innovation Critical Pilot Specific Technologies for Embedded Low-Energy Low-Cost Systems
- Task 8.6 - Common Arrowhead Integration Component Design and Implementation

WP 9 - Interoperability and Integrability Test Framework

This WP is meant to provide support to the other WPs in terms of a set of tests and demonstrations as well as tutorials and documentation on how to code and how to implement Arrowhead systems. The WP leader is BNearIT. Tasks:

- Task 9.1 - Arrowhead Application Support and Framework Test and Governance
- Task 9.2 - Definition of Compliance Classes to Arrowhead Specification
- Task 9.3 - Test Bed

WP 10 - Innovation and Standardization

The aim of this WP is to analyze and review the current business models related to SOA architecture in the current state of the art and thus provide the basic business intelligence necessary to build future offerings on the market. The WP leader is SKF. Tasks:

- Task 10.1 - Service Business Models Overview
- Task 10.2 - Business, Technology and Requirement Trend Screening
- Task 10.3 - Innovation Implementation Methodology
- Task 10.4 - Standardization

WP 11 - Business and Technology Dissemination

The main aim of this WP is to support the exploitation of results and solutions coming from any of the pilots in several domains such as industrial, societal and academical. The WP leader is VTT. Tasks:

- Task 11.1 - Business and Technology Dissemination Plan
- Task 11.2 - Business and Technology Publicity and Publishing
- Task 11.3 - Web, Twitter, YouTube

WP 12 - Project Management

As the name suggests, the aim of this WP is to enable a working methodology and a precisely scheduled interaction among the WPs, making them respecting the milestones, tracking the costs, performing a procedure on how to revise the plans. The WP leader is LTU. Tasks:

- Task 12.1 - Planning and Scheduling
- Task 12.2 - Progress and Cost Reporting
- Task 12.3 - Monitoring, Control and Quality Management
- Task 12.4 - Risk Management

The project presented in this dissertation is placed in Task 9.3, as it provides an implementation of a useful demo for many purposes. As the main topic covered is Electro Mobility, this project should have a parallelism with a real implementation provided by WP3, with which our team has a consistent collaboration. WP8 provided the Core Services (described in detail in section 3.1) and the core libraries used to implement the Arrowhead adapters, while Task 9.1 provided the necessary documentation. All the Work Packages shared their documentations, productions, demos and tests in a common SVN repository [35].

Chapter 2

The E-Mobility Simulator Platform

This section aims to describe the heart of the system presented in this dissertation: the UniBO E-Mobility Simulator Platform, a project that concretely started to gain shape in April 2012, when I started to write my Bachelor's degree dissertation [17]. Since then, many improvements have been performed and many people worked on the project, mainly Simone Rondelli [24], who reorganized the system using an efficient architecture design and implemented several new features.

2.1 A System Developed by UniBO

The system was initially a prototype which I personally developed as a contribution to the European project Internet of Energy [32], with the support from the UniBO team and the ARCES team, and leaded to the publication of an article [19] as their main contribution.

2.1.1 Overview

One of the main issues that the E-Mobility and smart energy production in general are, even nowadays, facing is given by the organization of the charging infrastructure. Internet of Energy initially proposed the concept of smart grid [18], which, through its connection to the Internet, would enable the production of an enormous set of informations. These informations would be used by end consumers to grant the energy balance between different zones, establish the price of energy in real-time, manage the charging station network and offer vehicle-to-grid services. Hence, the smart grid it is considered as a macro smart environment.

Since the European projects leaded by ARTEMIS aim to push the customers in the utilization of renewable energy sources and E-Mobility, the charging infrastructure shall constitute an efficient response to the marked demand. As stated and largely explained in my previous work [17], electrical vehicles can be recharged currently in two ways:

- Domestic plugs, which can provide a significantly limited power, thus a complete battery recharge takes in average up to 8 hours.
- Public plugs, the so-called charging stations, which can provide up to 50 kW in direct current and taking around 20 minutes to fully recharge an empty battery.

Thus, the main problem of the recharging infrastructure is identified in the duration of the recharging process, fostering the introduction of a reservation system in which users can book the charging station for a limited amount of time, avoiding this way the danger of long waiting queues. With the increasing development of the mobile industry, our project put a significant amount of effort in enabling the possibility to reserve the charging station directly from the user's mobile phone. With this purpose, UniBO and ARCES developed a demo system, which consists of a simulation of urban E-Mobility and recharging infrastructure, to give shape to the above mentioned system.

The need of a simulated E-Mobility environment derives from many questions coming from different domains: enterprises, vendors, researchers and end users. Therefore, here are reported the motivations and the purposes for which the E-Mobility Simulator Platform was developed and keeps being updated with many features according to various domains:

- Study the impact of a certain number of electrical vehicles on the charging infrastructure comprehending different customizable scenarios.
- Study the behavior of the electrical vehicles in presence of certain environment conditions, from the number and the availability of the charging stations to the effective autonomy and the energy consumption over time.
- Develop the real mobile client application which is "plugged" to one of the simulated vehicles to being able to test it. In this case the end user is really personified in one of the electrical vehicles in the simulation and can see his own vehicle moving on the map with significant accuracy, as well as performing reservations against the real service.
- Test and benchmark the underlying candidate architecture, Smart-M3, which is explained in detail further on in section 2.1.2.
- Provide the outer world with an unique interface, which would be the same as the one relying on the real system implementation, thanks to the Arrowhead adapters explained in chapters 5 and 6.

- Make possible to develop a pre-deployment “What If” mobile application, which aims to monitor and simulate the user’s real vehicle and give as an output the parameters that it would produce if it was electric.

Those are just few motivations, many of which have already been developed and demonstrated.

Entities

Most of the entities that characterize this system have been explained in detail in [17] and [24], thus here are reported briefly:

- **Electrical Vehicle (EV)**: the main entity of the project. They are divided into Fully Electrical Vehicles (FEV) and Plug-in Hybrid Electrical Vehicles (PHEV), even though in this project all EVs are considered to be FEVs, therefore their overall movement depends entirely on their battery capacity. EVs are part of a smart environment, as they can provide several information basically regarding their identification (RFID, User ID and so on), their position and their electrical parameters.

- **Grid Connection Point (GCP)**: this entity can be assumed as the closest to the meaning of charging station, as it represent geographically the connection point between the smart grid and the outer world. Each of those can host multiple charging structures, the EVSEs, described below, and, as a part of the smart environment, provides informations such as the position and the name.

- **Electrical Vehicle Supply Equipment (EVSE)**: it represents the other main entity of the whole project. It is defined as “The conductors, including the ungrounded, grounded, and equipment grounding conductors, the electric vehicle connectors, attachment plugs, and all other fittings, devices, power outlets or apparatuses installed specifically for the purpose of delivering energy from the premises wiring to the electric vehicle” [2]. Several types of EVSE are currently existing and installed, they can provide a socket either in alternative or in direct current, up to 50kW power and a current density up to 125A. As a part of a smart environment, it can provide several informations mainly regarding its electrical recharge profile, its current energy price and its availability over time.

2.1.2 Smart-M3

Within the project there has been an increasing need for an interoperability standard that grants its integrability and its scalability over time. In a system based on smart spaces and smart environments it is essential that the different domains and applications can rely to a single communication and cooperation architecture. The problem was raised [7] when the impossibility of coexistence among different protocols, such as UPnP for the home entertainment or the Apple ecosystem, as the number and the complexity of devices started to exponentially grow.

The solution was proposed mainly by Nokia and took the name of Smart-M3 [11][36]. M3 is a middleware platform, which realizes the interoperability of informations in a cross-domain, multi-vendor, multi-device, multi-platform way [20] and Smart-M3 is its first implementation proposed within the scope of the SOFIA project [37], one of the past ARTEMIS realizations for smart spaces. It has been subsequently used in other projects, such as Internet of Energy, and has been improved by ARCES over the years by adding more functionalities.

Smart-M3 operates through an architecture which promotes the loosely coupling between any producer (the sensor) and any consumer (the actuator or the end user) of the informations. The language chosen for the information exchange was RDF [42], an XML-based markup language, which encapsulates every information in triples ⟨subject, predicate, object⟩ and constitutes the basis for the Semantic Web paradigm. Indeed, data structured this way, in order to be given a semantic denotation, should follow rules dictated by an OWL [43] ontology, an RDF-based language which aims to give standardization to data. Although each application needs to be given its own ontology, several standard ontologies have been defined in order to enable interoperability around common use data.

The main entities composing the Smart-M3 architecture are reported below:

- **Semantic Information Broker (SIB)**: a SIB is a non structured database able to store data encoded in RDF triples. In such sense, it can be allocated in the set of graph databases, which are not designed to keep in memory a historian, but rather to describe the reality in a determined moment. Each single entity within the scope of the system shall use the SIB as a communication mediator, thus each message exchange happens through writing and reading triples from the SIB itself. For these reasons, each SIB has been provided with a set of several TCP/IP sockets which can be enabled in order to support tens of connections.

  Thanks to the hard work performed by the ARCES team in Bologna the SIB changed significantly over time, supporting currently the Redland Triplestore, many operations in SPARQL language rather than the obsolete WQL and high performances even while handling thousands of subscriptions at a time (see further).

- **Knowledge Processor (KP)**: a KP is any application or system able to communicate with the SIB through one or more of the designed operations. Often, it is necessary to write an adapter for a legacy existing system to support such feature. The communication occurs through a designed XML-RDF based protocol called SSAP (Smart Space Access Protocol) for which several libraries, commonly referred to as Knowledge Processor Interface (KPI), have been developed for some of the most common languages: Python, Java, C and so on so forth. As a previous additional contribution to the project I developed the integration for SPARQL queries in the C libraries denominated KPI Low [39] in order to facilitate the reachability of multiple informations [17].

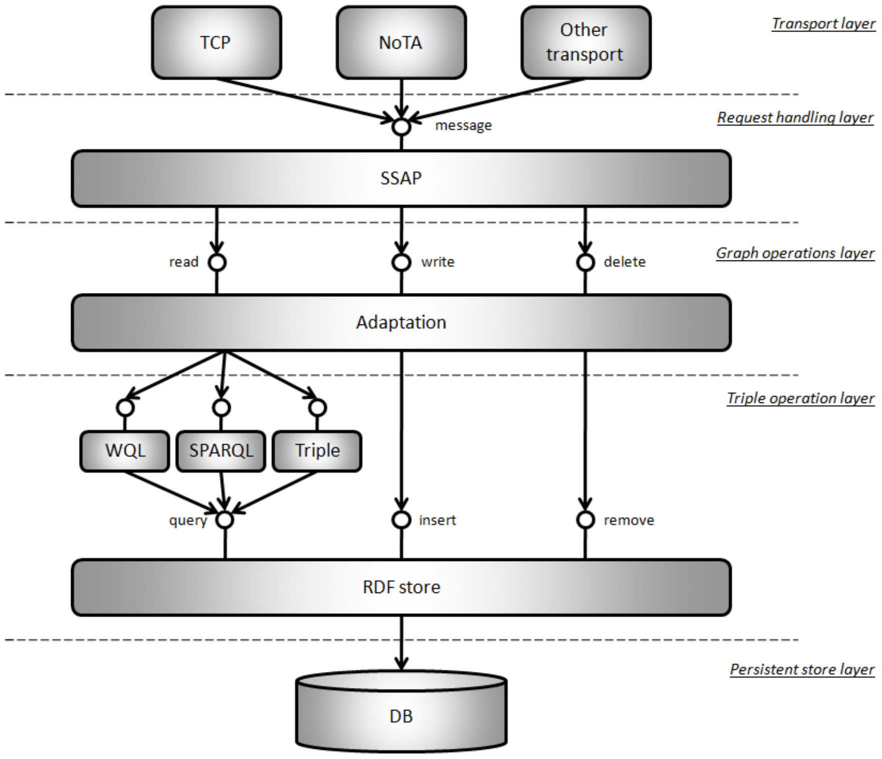


Figure 2.1: An overview

The SSAP Protocol

A KP can interface with a SIB in several ways, denoted and established as a set of precise operations encoded and encapsulated in a SSAP envelope. A KPI, by name, shall implement all of these operations to be able to grant the interoperability. Currently the following operations are available:

- **JOIN**: The operation without which no other operation can be performed. Since the protocol is session based, a KP cannot perform any other operation without initially join the smart space and providing its identity as well as its credentials. This operation is encapsulated in a SSAP envelope and does not use any other inner protocol.

- **LEAVE**: With this operation, a KP ceases the communication session with the SIB. After this, a KP cannot perform any other SSAP operation. This operation is encapsulated in a SSAP envelope and does not use any other inner protocol.

- **INSERT**: With this operation, a KP inserts in the SIB a graph of determined triples. This operation is currently supported in RDF-M3 and SPARQL formats.

- **DELETE**: With this operation, a KP atomically specifies a graph of triples that has to be deleted from the Smart Space. This operation is currently supported in RDF-M3 and SPARQL formats.

- **UPDATE**: With this operation, a KP atomically specifies a graph of triple that it intends to update and a graph of triples which are the replacement to the others. The SIB indeed performs initially a DELETE operation and subsequently an INSERT. This operation is currently supported in the format RDF-M3.

- **QUERY**: With this operation, the KP atomically fetches a number of triples that satisfy the parameters given. The operation is supported in RDF-M3, with which a subset of subject, predicate or object can be specified. As an alternative, widely used within the scope of the project, the operation can support SPARQL language. SPARQL is a language in some ways specular to SQL on non-relational databases, thus it support insert operations, queries, updates and deletions. Although SPARQL operations are always encapsulated in a SSAP QUERY envelope, they can all be performed.

- **SUBSCRIBE**: The subscription is one of the most important features of the Smart-M3 architecture. A KP can subscribe to one or more triples (one or more of its parts can be parametrized); with this operation the SIB notifies the KP whenever a triple belonging to the specified pattern is inserted, deleted or updated. Logically, on the other hand, the KP must maintain an open socket listening for the notifications. Currently RDF-M3 and SPARQL languages are supported.

- **UNSUBSCRIBE**: With this operation, a KP atomically disables any subscription with the given parameters.

2.1.3 Simulation Platform

As largely explained in Simone Rondelli's dissertation [24], many technologies have been combined to obtain an efficient basis for the development of the system. In this case our choice was an interaction between diverse well-known simulator platforms, allowing the developer to rely both on an efficient and basic simulation of urban mobility and on an efficient simulation development environment. Below are described in sufficient detail those systems.

SUMO - Simulation of Urban MObility

SUMO is an open-source, multi-platform simulator of urban mobility written in C++ supported by the Institute of Transportation Systems at the German Aerospace Center [12]. It simulates big traffic networks in which every vehicle is designed in a microscopic way, thus it has its own characteristics, its own itinerary and its own behavior. Every input to the simulation is parametric and constituted by an XML document, for instance the vehicle routes, the map, the buildings and so on so forth. It supports, when needed, a powerful and interactive GUI and a set of tools aiming to help the user generating the XML parametric files. Those tools are Netconvert (able to convert a map from a standard format such as OpenStreetMap), Duarouter (able to generate the vehicle routes) and Polyconvert (able to generate the polygons such as the buildings imported from other formats). The interaction with SUMO is given by a module called TraCI (Traffic Controller Interface), which provides several operations callable from the extern according to a client/server protocol [8].

OMNeT++

OMNeT++ is an open source environment for developing discrete events simulations, distributed together with an Eclipse-based IDE [5]. Despite such environment has been designed purely for network simulation, its flexible nature made possible to use it for a wide variety of different purposes. It is based principally on the interaction among modules through message passing, which can occur together with any structured format. Such modules interact using gateways and can be compound with other modules, each performing a different task.

One of the most powerful tools provided by OMNeT++ is the data analysis. The developer can ask to the system to register any of the vectorial or scalar values of interest during the simulation and to automatically plot a histogram, a Cartesian graph, a bar graph, a linear graph and so on.

Veins

Veins is an Open Source framework for the Inter-Vehicular Communication (IVC) [13] based on OMNeT++ and MiXiM [9], a framework itself which simulates adhoc networks, wireless sensors networks and vehicular networks [14]. Veins communicates continuously with SUMO using the TraCI module and establishes a parallelism between the vehicles in SUMO and the modules in OMNeT++. For our purposes, we removed from the modules the submodules regarding the network protocols (802.11 and ARP) and added several other modules, explained in section 2.4. Figure 2.2 shows the interaction between the modules.

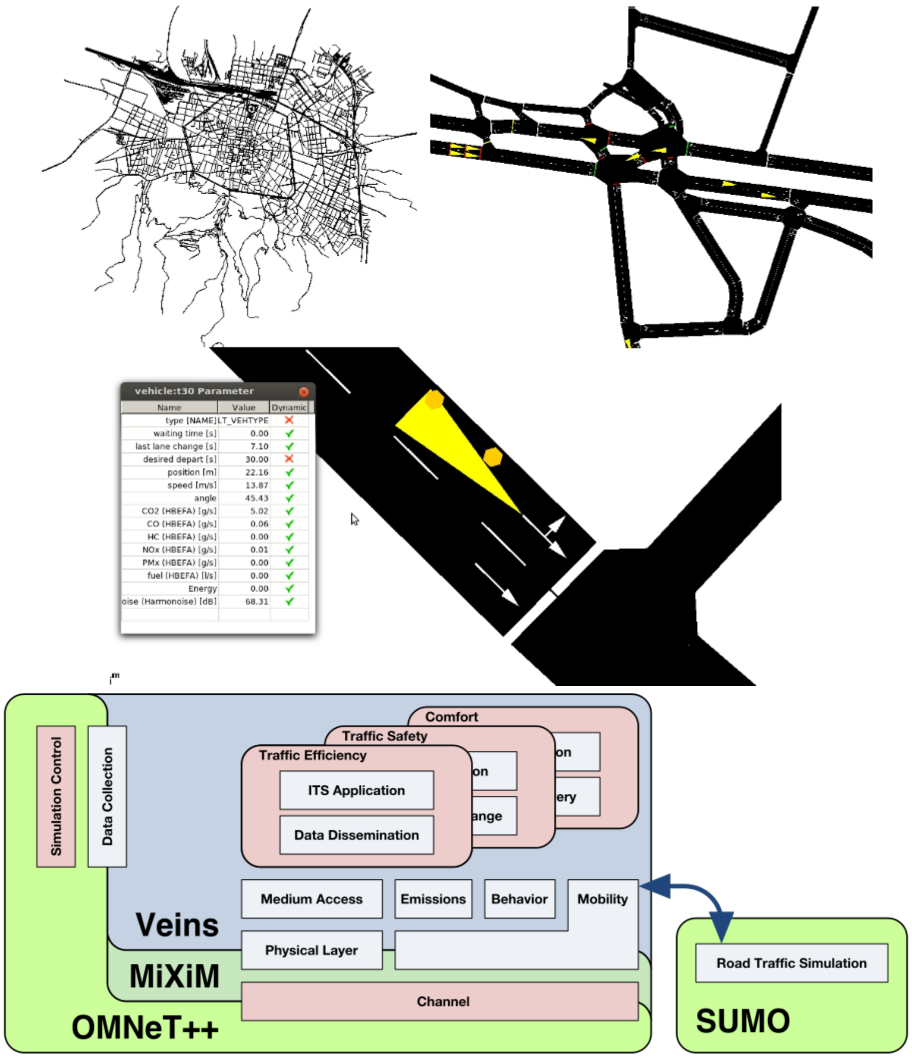
2.2 Project Structure

This section shows how components are acting within the E-Mobility Simulator Platform and how we reached and achieved the goal of simulating each interactions among the modules.

2.2.1 Internal Architecture

According to the Smart-M3 paradigm, the architecture is composed by a set of components that communicates with each other through the data written and read from one or more SIB. As the relevant data coming from the smart space formed by the simulated environment and the city service is regarding both vehicles and charging stations, we decided to physically split the storage system in two different SIB. This choice refers primarily to the fact that some of this data (generally the one regarding vehicles) should not be public in a real world implementation, while data regarding charging stations and reservations must definitely be reachable by any user whatsoever.

Figure 2.3 shows the overall architecture. It is evident how active components (here identified by circles denoting single processes) are all acting in full as KPs, in fact not a single direct communication among them is present. Although the architecture here represented denotes only how the legacy system has been designed before being integrated in the Arrowhead framework, it is important to point out clearly that any add-on module or adapter subsequently integrated shoud respect this architectural style. Therefore, it is expected that the Arrowhead service provider presented in chapter 5 shall fetch the information needed only from the


vehicle:t30 Parameter

Name	Value	Dynamic
type [NAME]	LT_VEHTYPE	✗
waiting time [s]	0.00	✓
last lane change [s]	7.10	✓
desired depart [s]	30.00	✗
position [m]	22.16	✓
speed [m/s]	13.87	✓
angle	45.43	✓
CO2 (HBEFA) [g/s]	5.02	✓
CO (HBEFA) [g/s]	0.06	✓
HC (HBEFA) [g/s]	0.00	✓
NOx (HBEFA) [g/s]	0.01	✓
PMx (HBEFA) [g/s]	0.00	✓
fuel (HBEFA) [l/s]	0.00	✓
Energy	0.00	✓
oise (Harmonoise) [dB]	68.31	✓

Figure 2.2: Overview of the SUMO GUI (above) and the architecture of Veins (below)

SIB.

Below are described more in detail the entities of the system.

City SIB

The City Service Information Broker is one of the main entities of the overall project. It not only constitutes the bottleneck through which almost all the communications take place, but indeed it acts also as a gateway to the outer world.

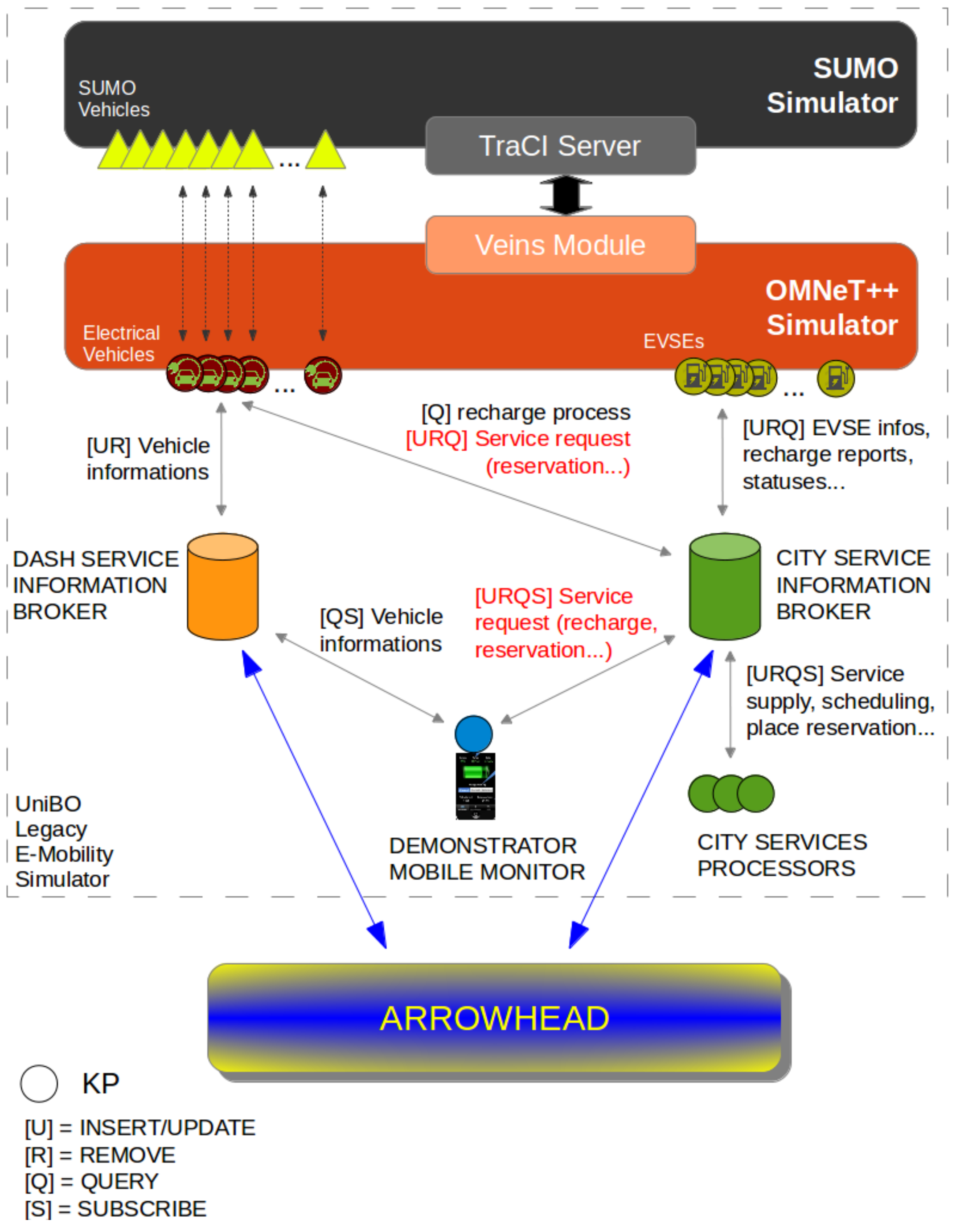

SUMO Simulator
SUMO Vehicles
TraCI Server
Veins Module
OMNeT++ Simulator
Electrical Vehicles
EVSEs
[UR] Vehicle informations
[Q] recharge process
[URQ] Service request (reservation...)
[URQ] EVSE infos, recharge reports, statuses...
DASH SERVICE INFORMATION BROKER
CITY SERVICE INFORMATION BROKER
[QS] Vehicle informations
[URQS] Service request (recharge, reservation...)
[URQS] Service supply, scheduling, place reservation...
UniBO Legacy E-Mobility Simulator
DEMONSTRATOR MOBILE MONITOR
CITY SERVICES PROCESSORS
ARROWHEAD
KP
[U] = INSERT/UPDATE
[R] = REMOVE
[Q] = QUERY
[S] = SUBSCRIBE

Figure 2.3: Architecture of the whole legacy E-Mobility Simulator. In red are identified the operation no longer required, while in blue the interactions with the outer world, still respecting the Smart-M3 architecture.

It is meant to store all the data supposed to be available in a real implementation (which would probably consist of a set of City SIBs or, more probably, an ad-hoc cloud), starting from the collection of parameters coming from the whole set of charging stations (EVSEs and GCPs). It is also the mediator between the vehicles, the City Service and the EVSEs within the scope of the reservation process (as stated later on). Furthermore it stores the ontology, described in detail in section 2.3, which must be carefully followed in order to go standard and make the whole system working properly.

Dash SIB

The Dash Service Information Broker is a secondary SIB, but essential to retrieve informations about vehicles. It was necessary to split the storage in two separate concepts, as the fact that vehicle informations (such as GPS position or State of Charge) are public would probably not meet the requirements in a real implementation. Keeping the vehicle informations separated allows this system to be flexible and as parallel as possible with the reality. In particular, all the information stored in the Dash SIB consist merely of the whole set of parameters provided by the electrical vehicles, together with a duplicate of the ontology.

Electrical Vehicle KP

The Electrical Vehicle Knowledge Processor is responsible for governing the whole life cycle of the vehicle. Its functioning is explained in further detail in subsection 2.4.1. In general it manages the operations of driving, deciding the route based on some sort of predefined behavior, calculating the battery consumption, monitoring the battery and its electrical parameters, reporting constantly its position, requesting a reservation when the state of charge is low, choosing a reservation spot among the ones offered, going to the designed charging station, calculating the energy acquired during the charging process and unplugging the vehicle when the battery is full. All its data during all these operations are reported for testing and monitoring purposes on the Dash SIB, except for the reservation request and confirmations, which have to be public.

This KP was initially developed by me in 2012 [17], then refactored by Luca Bedogni and Simone Rondelli [19][24] and again modified by me in the scope of the present project.

EVSE KP

The EVSE Knowledge Processor is responsible, specularly to the previous, for governing the whole life cycle of the EVSE.

Basically it has a double purpose: it posts constantly on the City SIB any variation in its electrical parameters to keep them monitored from the extern and reads constantly from the SIB any inducted variation in those same parameters for testing purposes and it keeps the recharging process monitored, even if, in our implementation, the charging progress is published on the City SIB by the vehicle KP.

This KP was developed by Simone Rondelli in 2014 [24], then modified by me in the scope of the project.

City Service KP

The City Service KP acts as an extern server against the simulator. In fact it runs separately and it can be considered a prototype of the real implementation rather than a simulation. The first and foremost task it performs is the handling of the whole reservation process (described further in subsection 2.2.2). To achieve this result it must exploit the subscription mechanism of the SIB, thus subscribing to any variation, insertion or deletion of charging request and reservation confirmation. Moreover, in the scope of the simulation, it provides the SIB with the initial and fixed informations about all the GCPs and EVSEs, retrieved from an XML configuration file, indeed acting as an initializer.

A first prototype was developed by me in 2012 [17], then completely redesigned by Simone Rondelli [24] to achieve a better scalability and usability, then again modified with some more features in the scope of this project.

Demo Mobile Monitor KP

The Mobile Monitor KP was developed in the first scope of the project and is currently obsolete. Its purpose is to simulate the real application running on the end user's mobile phone. With the aid of a third-party technology, such as Blue&Me, relying on the Bluetooth protocol, the application must be in constant communication with the EV's control unit and fed by its parameters [24]. In the scope of the simulation, the data are fetched directly from the Dash SIB using again the subscription method, therefore the application is updated every time a pre-designated vehicle changes it parameters on the SIB. Basically, the mobile application works exactly as if the user was driving one of the simulated vehicles; it can see its position on the map, it can switch to a screen showing all its electrical parameters (especially the State of Charge) and it can perform reservation requests just as if it was driving the real vehicle.

A first prototype was developed as proof of concept by me in 2012 [17], then completely rebuilt by Simone Rondelli [24] as his first contribution to the project. Subsequently several people gave contribution to the project as well until July

2014, when it became obsolete.

2.2.2 The Recharging Reservation Process

The most powerful feature added to the simulation was the recharging process and the reservation mechanism. Although this mechanism had to be redefined in the scope of the Arrowhead integration, it is useful to report here briefly how it has been implemented before, as this implementation is meant to persist in the standalone simulation as well as for the reservation performed automatically by the non-monitored vehicles. The reservation process is due to the exchange of particular sets of triples, described in subsection 2.3, and is here reported from a rather abstract point of view. Further details can be found in [17] and [24].

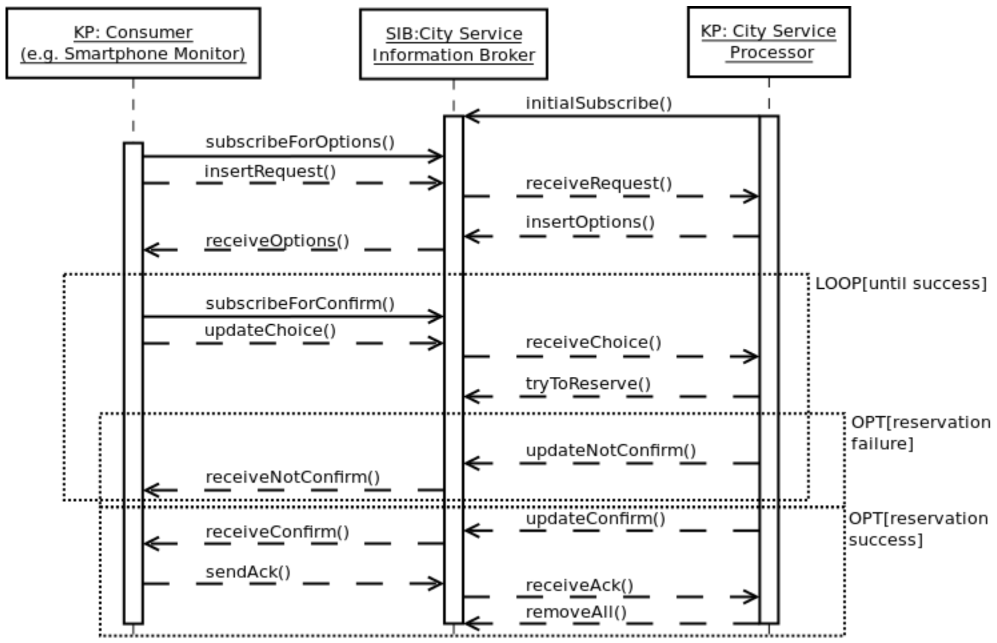


Figure 2.4: Legacy reservation process handled by the City Service, the City SIB and a Client.

The process is articulated in several phases, reported in the list below as well as in figure 2.4, and all of them are mediated by the City SIB:

1. **Recharge Request**: the vehicle's user places a reservation request, nominally `ioe:ChargeRequest` in subsection 2.3, providing its position, its state of charge and its preferred time.

2. **Recharge Response**: the City Service responds with a response, nominally `ioe:ChargeResponse` in subsection 2.3, providing, after analyzing the status and the Reservation queue of each EVSE, a set of options, characterized by a price and a time slice.

3. **Confirm by User**: the vehicle's user updates one of the options marking as true the flag `ioe:ConfirmByUser`, claiming that he wants to place a reservation corresponding to that option.

4. **Confirm by System**: the City Service checks if the spot is still available, in such case marks as true the flag `ioe:ConfirmBySystem` for that option. On the other hand, if the spot is not available, the system provides another response and goes back to phase 2.

5. **Acknowledgement by User**: the vehicle's user marks as true the flag `ioe:AckByUser` of that same option. This often is done below the actual control of the user, just to check if the client did not crash in the meantime.

6. **Reservation Insertion**: once received the acknowledgement, the City Service finally places a reservation for that vehicle on the SIB.

The process of deleting a reservation is rather simple, thus it will not be reported here in detail. The user simply asks for a reservation retire and the City Service executes it and subsequently notifies the user with an acknowledgement. The whole communication is again mediated by the City SIB, for further details see [24].

2.3 Ontology

In a project that involves a consistent number of entities, interoperability is the key for a clean interaction. In particular, since the present system encodes every data in RDF triples, it is necessary to establish a standard which governs the format of every communication, especially when the data to be represented is heterogeneous. OWL is a global format, used within the scope of Semantic Web and Linked Data, defining RDF/XML standard data structures and it has been chosen for the data representation within the project. The vast majority of the information used is not already present in any existing OWL ontology whatsoever, thus only few references to external ontologies have been used.

The ontology has been widely modified in the scope of the integration of the legacy system within the Arrowhead framework, thus here is reported the updated version, which is used currently: *ioe-ontology_v1.7.2.owl*. All the ontology triples, encoded in RDF/XML format, are uploaded physically on each SIB during its initialization as well as available in the same format as an OWL document on the project repository on bitbucket: `https://bitbucket.org/InternetOfEnergy/internet-of-energy`. Due to this shrewdness, the ontology will be available in full to each developer willing to use our customized version of the Smart-M3 architecture.

Each class, instance or property here represented is intended to be indicated with the ontology's namespace as a prefix every time is referenced anywhere in the project. The complete prefix is `http://www.m3.com/2012/05/m3/ioe-ontology.owl#`.

2.3.1 Entity Classes

All the OWL classes reported here are subclass of `owl:Thing` (unless differently specified) and denote a physical entity of the system. Although the Object properties and Datatype properties are more numerous than the ones represented here, we try to give all the elements necessary to understand how to use the ontology entities.

- **Vehicle**: is the main entity of the project. It represents an electrical vehicle (which can be fully or hybrid).

hasBatteryData	BatteryData	Link to the battery data.
hasGPSData	GPSData	Link to the position data.
hasIdentification-Data	IdentificationData	Link to the user identification data.
hasReservation	Reservation	Link to its active reservation.
hasManufacturer	data:string	Manufacturer's name.
hasModel	data:string	Model's name.

- **GridConnectionPoint**: one of the main entities of the project. It represents a charging station (which can include one or more charging spots, i.e. EVSE).

hasConnectedEV	Vehicle	One or more links to the vehicles connected.

hasGPSData	GPSData	Link to the position data.
hasIdentification-Data	IdentificationData	Link to the RFID identification data.
hasEVSE	EVSE	One or more links to the EVSEs hosted.
hasMaxPower	PowerData	Link to the maximum output power.
hasName	data:string	GCP's fiendly name.
supportsV2G	data:boolean	If supports Vehicle-to-grid operations.

- **EVSE**: one of the main entities of the project. It represents an EVSE.

hasAvailability	Availability	Link to the EVSE's availability status.
hasFaultCode	FaultCode	Link to the EVSE's fault code status.
hasChargingStatus	ChargingStatus	Link to the EVSE's charging status.
hasChargeProfile	ChargeProfile	Link to a set of electrical parameters.
hasChargeProgress	Recharge	Link to the profile of the ongoing recharge.
hasLastCharged-Energy	Recharge	Link to the profile of the last completed recharge.
hasConnector	Connector	Link to the connector's parameters.
hasGridConnection-Point	GridConnectionPoint	Link to the GCP hosting the EVSE.
hasMaxCurrent-DensityIn	CurrentData	Link to the max current density in input.
hasMaxCurrent-DensityOut	CurrentData	Link to the max current density in output.
hasMaxEnergy-Capability	EnergyData	Link to the maximum amount of energy rechargeble at a time.
hasMaxPower	PowerData	Link to the maximum output power.

hasMaxVoltage	VoltageData	Link to the voltage data.
hasReservationList	ReservationList	Link to the EVSE's list of reservations.
hasName	data:string	EVSE's fiendly name.
hasEvseIdentifier	data:string	EVSE's fiendly identifier.

- **Connector**: it represents an EVSE's connector (which can rely on different standards).

hasConnectorType	data:string	Name of the connector's standard.
hasStatus	data:string	Status of the connector (not used at the moment).

- **ChargeProfile**: abstract entity collecting electrical parameters belonging to the EVSE.

hasCurrent-DensityIn	CurrentData	Link to the instant current density in input.
hasCurrent-DensityOut	CurrentData	Link to the instant current density in output.
hasPower	PowerData	Link to the current output power.
hasVoltage	VoltageData	Link to the voltage data.
hasPrice	PriceData	Price of the energy per KWh (conditioned by external agents such as FlexOffer).

- **EVSEStatus**: identifies the status of an EVSE. It comprehends the OWL subclasses FaultCode, Availability and ChargingStatus. FaultCode can be currently: Enabled, Fault, Abnormal, Verification. Availability can be currently: Available, CheckFault, Plugged, Reserved. ChargingStatus can be currently: Unplugged, Recharging, StartCharging, StopCharging. Both ChargingStatus and FaultCode condition the Availability characteristic.

hasSignature	data:string	The status itself.

- **IdentificationType**: identifies a string that somehow describes data belonging to the caller.

hasSignature	data:string	The description itself.

- **Person**: identifies a person involved in the system (a vehicle's driver).

hasName	data:string	The person's friendly name.
hasUserIdentifier	data:string	The person's friendly username.

- **Zone**: identifies a square zone, denoted by its south-east point and its nort-west point.

hasSouthEastPoint	GPSData	Location of the south-east point.
hasNorthWestPoint	GPSData	Location of the north-west point.
hasIdentification-Data	IdentificationData	The zone's identification.

- **UnitOfMeasure**: identifies a string that denotes the unit of measure used in a measurement.

hasSignature	data:string	The unit of measure signature.

- **Currency**: identifies a string that denotes the currency used in a payment.

hasSignature	data:string	The currency signature.

- **ChargeRequest**: identifies a charge request inserted by a user in the SIB, waiting to be processed by the city service.

hasRequestingVehicle	Vehicle	Link to the requesting vehicle.
hasRequestingUser	Person	Link to the requesting user.
hasRequestedEnergy	EnergyData	Link to the amount of energy requested.
hasSpatialRange	SpatialRangeData	Center and radius of the area in which the request is intended to be valid.
hasTimeInterval	TimeIntervalData	Span of time in which the request is intended to be valid.
allowBidirectionality	data:boolean	Allow vehicle-to-grid operations.

- **ChargeResponse**: identifies a charge response inserted by the city service in the SIB.

hasRelatedRequest	ChargeRequest	Link to the request for which this is the response.
hasChargeOption	ChargeOption	Link to the options presented in this response to the user.

- **ChargeOption**: identifies a charge option inserted by the city service as part of a response.

hasRequestingVehicle	Vehicle	Link to the requesting vehicle.
hasGCPPosition	GPSData	Link to the position of the charging station.
optionHasEVSE	EVSE	Link to the designed EVSE.
hasTimeInterval	TimeIntervalData	Span of time in which the option is intended to be offered.
hasTotalPrice	PriceData	Total price of the operation.
hasChargeProfile	ChargeProfile	Link to the EVSE's charge profile offered.
allowBidirectionality	data:boolean	Allow vehicle-to-grid operations.

confirmByUser	data:boolean	The user selected this option.
confirmBySystem	data:boolean	The system inserted a reservation for this option.
ackByUser	data:boolean	The user knows about the reservation he or she just performed.

- **ReservationList**: identifies a reservation list belonging to a specific EVSE.

hasReservation	Reservation	Link to the reservations in this list

- **Reservation**: identifies a finalized reservation, inserted by the city service.

reservedByVehicle	Vehicle	Link to the reserved vehicle.
reservationHasUser	Person	Link to the user who performed the reservation.
hasEVSE	EVSE	Link to the EVSE for which the reservation is valid.
hasPrice	PriceData	Total cost of the operation.
hasTimeInterval	TimeIntervalData	Span of time in which the reservation is intended to be valid.
idBidirectional	data:boolean	True if the reservation allows vehicle-to-grid operations.

- **Recharge**: identifies a recharge process, inserted by the EVSE.

reservedByVehicle	Vehicle	Link to the recharging vehicle.
hasUser	Person	Link to the user who booked the recharge.
rechargingOnEVSE	EVSE	Link to the EVSE which is recharging the vehicle.
hasTimeInterval	TimeIntervalData	Span of time in which the recharge is effectively taking place.

hasEnergyData	EnergyData	Amount of energy recharged so far.

- **ReservationRetire**: identifies a reservation retire request, inserted by the user.

retiredReservation	Reservation	Link to the retired reservation.
retiredByUser	Person	Link to the user who performed the retire.

2.3.2 Data Classes

All the OWL classes reported here are intended to represent measured data within the scope of the project. They are approximately all subclasses of `http://www.m3.com/2012/05/m3/ioe-ontology.owl#Data` and, given their name, they can be all monitored by any application connected directly so the SIB. Most of them are monitored by the Arrowhead adapters as well.

In this first list we report all the data represented with the two properties `hasValue` and `hasUnitOfMeasure`, respectively referring to an object of type `data:double` and `UnitOfMeasure`, to avoid being too verbose. The scheme of the following table is: class name, unit of measure and short description.

ChargeData	kWh	Capacity data used in recharge processes.
CurrentData	A	Current density.
EnergyData	kWh	Energy data used in recharge and discharge processes.
PowerData	kW	Electrical Power.
TemperatureData	C°	Temperature, used to monitor the battery.
VoltageData	V	Electrical voltage.

There are several other data measurements which are represented in a slightly different structure; they are listed below:

- **BatteryData**: it is the most important data record of the project, which stores all the electrical parameters (nominal and variable) of a vehicle battery.

hasCapacity	ChargeData	Capacity data used in recharge processes.
hasCurrent-DensityIn	CurrentData	Current density in input.
hasCurrent-DensityOut	CurrentData	Current density in output.
hasMaxCurrent-DensityIn	CurrentData	Maximum current density in input.
hasMaxCurrent-DensityOut	CurrentData	Maximum current density in output.
hasNominal-Temperature	TemperatureData	Nominal temperature of the battery.
hasTemperature	TemperatureData	Current temperature of the battery.
hasPower	PowerData	Current power consumed by the battery.
hasVoltage	VoltageData	Nominal Voltage of the Battery.
hasStateOfCharge	ChargeData	Important, the amount of charge left in the battery.
hasStateOfHealth	data:double	The health of the battery, from 0 to 1.
hasManufacturer	data:string	Name of the battery's manufacturer.

- **IdentificationData**: It represents the identification of an entity, may vary according to the entity involved.

hasIdentification-Type	IdentificationType	The type of identification used by the entity.
hasCode	data:string	Code representing the entity, following a certain standard.

- **GPSData**: It represents the position of an object in terms of GPS coordinates. It is a subclass of LocationData.

hasGPSLatitude	data:double	Latitude of the object.
hasGPSLongitude	data:double	Longitude of the object.

- **SpatialRangeData**: It represents a circular area, often centered by GPS coordinates.

hasRadiusKm	data:double	Length of the circle's radius in Kilometers.

- **TimenIntervalData**: It represents a span of time, using as unit of measure the absolute time, so the number of milliseconds since January 1st 1970.

hasStartingTime-Millisec	data:double	Beginning of the time span.
hasEndingTime-Millisec	data:double	Ending of the time span.

2.4 Interaction in the Scenario

In this section the main entities implemented in the simulator are presented. The core functionalities of the whole simulation are developed within the OMNeT++ workstation, a powerful Eclipse-like IDE, in C++. The entities which take part in the simulation process are the following (not all of them are physical NED modules):

- The Car, the most important module, which is permanently tied to its counterpart in the parallel SUMO simulation. It is compound by different modules, each in charge for governing a particular set of actions that the driver (or the car) can perform: CarLogic (the main module aiming to monitor the car's status and its global decisions), DriverBehaviour (a module which performs decisions based on a set of behavioral parameters), Battery (the module which controls the battery parameters), CarMessage (the module which represents the maintenance of volatile data during the transition from an instant to another within the simulation).

- The GCP, which holds the static data regarding the Grid Connection Point.
- The EVSE, an active entity in the scenario, though not a NED module, which keeps track of its status and of the vehicles recharging and queuing at it.
- The CityService, which is not the external city service in charge of replying to the reservation request, but it aims to slice the simulation in instants, keep track of the number of vehicles and their status and instantiate some global properties (explained in subsection 2.4.2).
- The Synchronizer, which is an unique entity aiming to synchronize the simulation to any other concurrent simulation that can interact with it (services for energy pricing, Smart Grid balancing and so on).
- The GcpController, which is an unique entity that acts as an bottleneck to initialize GCPs and EVSEs first (which are pre-determined in an XML configuration file) and access to any of them later.
- The SibController, an object present in every Car module, which manages all its connection to both the SIBs.

2.4.1 The Electrical Vehicle's Lifecycle

The main point of interest of the simulator is how the electrical vehicles behave, mostly because everything that change within the simulator is due to the actions performed by them. Vehicles are spawned in the simulator according to a parametric frequency and they are considered electrical due to another parametric frequency. When this happens, if the vehicle is not considered electrical (red vehicle in figure 2.5), its OMNeT++ module is erased and it is not considered (although it will still take part in the simulation as a part of the traffic and it will run along its pre-determined path). If, otherwise, the vehicle is electrical it will load all its nominal and temporary data, as well as its battery parameters, in the dash SIB and it will be assigned the status **DRIVING**. Due to the Sumo configuration file, the vehicle has a specific route assigned to it, although the process of battery discharging is running in parallel. The process itself has been developed according to the battery specifications given by Siemens [17], followed by many integrations that included the inertial auto-recharge and the slope data, as well as a better refactoring of the equation [24]. Each car, to keep in memory

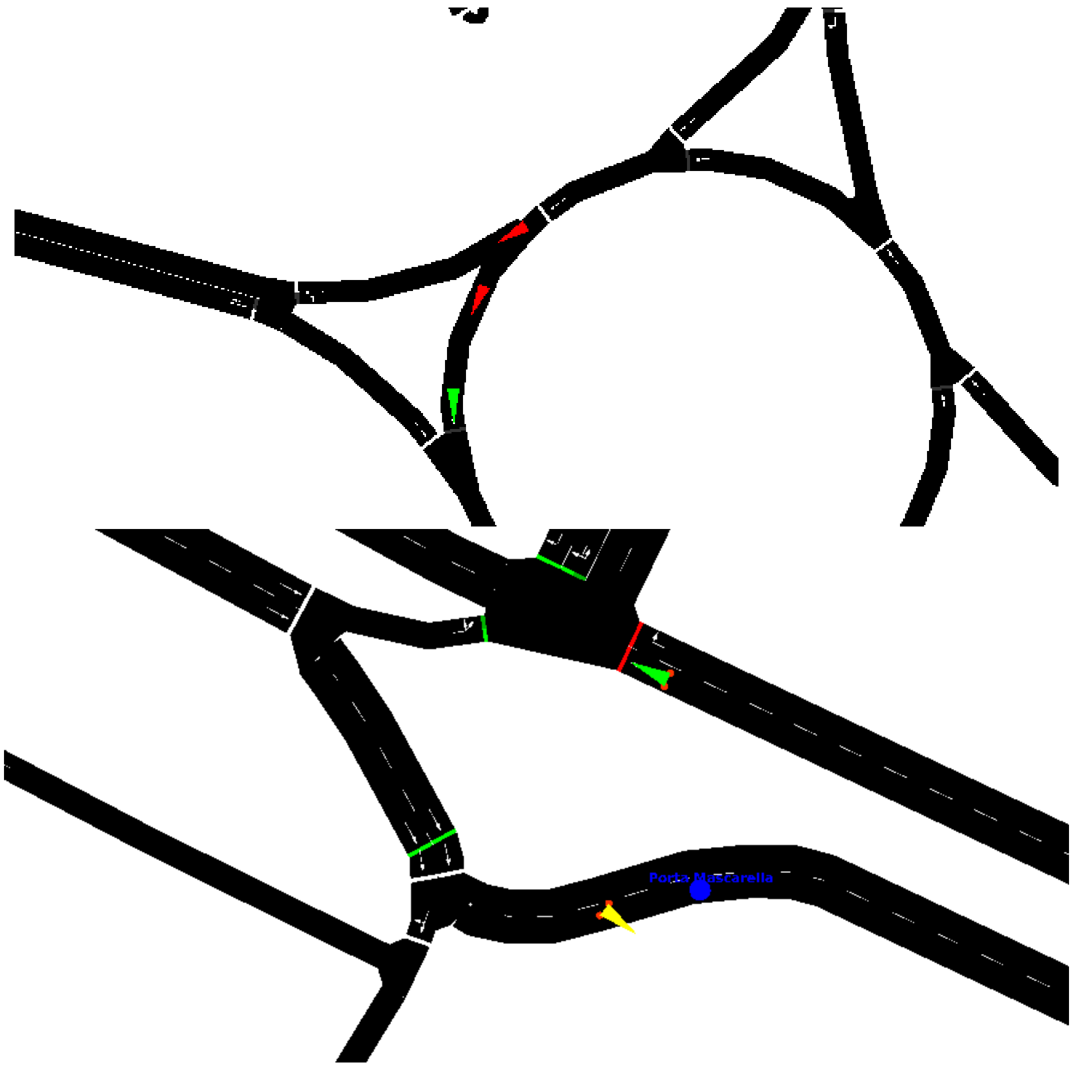


Figure 2.5: Vehicles in our system seen through the Sumo GUI. A green vehicle is electrical, a red vehicle is fueled and a yellow vehicle is an electrical vehicle either parked or recharging.

its volatile data, envelopes it in a “self message”, scheduled for the next discrete event (normally those discrete events occur once in a second). When the parameter “State of Charge” (SoC), the most important and relevant parameter of the battery, is perceived lower than a certain percentage (which is parametric) the car’s SibController sends a charge request to the City SIB and the car switches to the state **WAITING RESPONSE**. At this point, the vehicle keeps polling the City SIB for the charge response until it receives it. When the charge response is fetched, the vehicle chooses randomly one of the options and switches to the

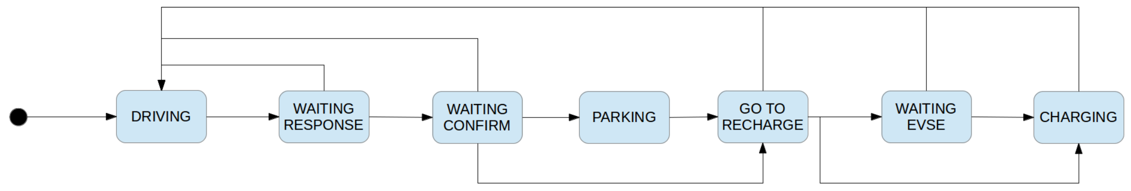


Figure 2.6: Overview of the vehicle activity flow.

state **WAITING CONFIRM**. If the option is not confirmed the car switches back to the previous state and cycles the operation, while, if the option is confirmed, the vehicle calculates exactly how much time would require to reach the chosen charging station and compares it with the reservation starting time. If it realizes that it would get there before the reservation is valid it assumes the status **PARKED**, parks along the road, if possible, and schedules a "self wakeup message" after $(reservationTime - (currentTime + timeToReachStation)$ and when the message arrives it assumes the state **GO TO RECHARGE**. If the vehicle realizes that there's no need to wait, then it switches immediately to the state **GO TO RECHARGE** and, as soon as this is the vehicle state, it changes its route with the position of the chosen charging station as a destination. When the vehicle arrives at the charging station, it parks and, if there is no queue, it switches to the state **CHARGING**, while, if all the EVSEs are occupied it performs a choice based on the rules stated in the module DriverBehavior. This choice is determined by a simulated mind status similar to the driver's anxiety or impatience and it is based on the estimated waiting time in line for a free EVSE. If the driver decides to wait, then the car switches to the status **WAITING FOR EVSE** and it will be waken up by the EVSE itself when it will be free. If, on the other hand, the driver decides not to wait it switches to the status DRIVING and tries another charge request, so the process starts over again. Normally, due to the reservation protocol (i.e. is not possible to reserve the same spot twice) the situation in which the vehicle waits for a free EVSE should not happen, though several modifiers (decided previously in the configuration) may interact deeply with the simulation itself.

Main Changes

One of the main changes from the older version of the simulator is the presence of status flags about the EVSEs and the concept of fault, which has been introduced as a requirement by WP3. Before these new features, the only status information retrievable about the charging station was if the EVSE was recharging a vehicle or not.
The new EVSE status is compound and hierarchical. It is compound by three main

objects: the availability, the charging status and the fault code, which interact among themselves in the following way:

1. The availability (Available, CheckFault, Plugged, Reserved) may never be set from the extern, it is only conditioned by other statuses.
2. The charging status (Unplugged, StartCharging, StopCharging, Recharging) affects the availability, in particular, if is set on a value different than Unplugged, the availability is set on Plugged.
3. The fault code (Enabled, Fault, Abnormal, Verification) affects both the availability and the charging status, in particular, if is set on a value different than Enabled, the charging status is set on Unplugged and the availability on CheckFault.

Whenever an EVSE faults, it may involve vehicles as well. If the EVSE was recharging, each vehicle in CHARGING or WAITING FOR EVSE status against that EVSE will immediately assume the DRIVING status and try to find some other EVSE by reservation. As each vehicle keeps monitoring the EVSE where reserved the spot, if the faulted EVSE was reserved each vehicle reserving that EVSE will generate a notification which triggers the deletion of its reservation for that EVSE and forces the vehicle to perform another request elsewhere.
Several other smaller improvements were performed on the simulator.

2.4.2 Parameters

Whenever a new launch configuration is decided, several parameters may be instantiated, making possible different case studies. Those configuration are stated in the `omnetpp.ini` file in the root directory of the simulator. Some of the most relevant parameters are stated below:

- **electricalVehicleFreq**: this parameter indicates (in percentage) the frequency for which any spawned vehicle would be electrical.
- **maxElectricalVeh**: this parameter indicates what is the maximum number of electrical vehicles at a time (when the maximum amount is reached all the generated vehicles are fueled).
- **reservationEnabled**: one of the most important parameters of the project. If false, the reservation mechanism would no be used at all and the vehicles would start to go to recharge to the closest charging station eventually generating waiting lines (used mainly in user behavior case studies).

- **EVSEFaultProb**: this parameter expresses, in percentage, how likely is an EVSE to be spawned in a faulty condition.
- **writeCarStatusOnSib**: this parameter, if set to true, enables the vehicle to update the Dash SIB with its status. It is clearly just for monitoring and testing purposes and it does not have any real counterpart in a real scenario, as the vehicle's status should not be publicly reachable. Furthermore, it considerably slows down the simulator's performances as it requires two operations (insertion and deletion) against a single SIB each time slice multiplied by the number of active electrical vehicles.
- **threshold**: this parameter indicates (with a float normalized value from 0 to 1) the state of charge threshold below which a vehicle will automatically perform a charge request.

There are several other parameters in the Battery module as well as in the Driver-Behavior module to customize some battery and behavioral parameters.

2.5 Some Results

In this section we report briefly some past results obtained by this project. Those are important to be described as they clearly state that we reached some of the goals stated in section 2.1.1, in particular how a certain number of electrical vehicles impacts on a charging infrastructure with or without the reservation mechanism. The complete description can be found in [24].

2.5.1 Vehicle Consumption

The figure 2.7 shows the state of charge of an electrical vehicle over the whole simulation. Time is displayed on the x axis and state of charge on the y axis. It can be noticed how the vehicle can reach almost every time a free EVSE before the battery runs out of energy. The threshold is set differently for each vehicle; it can be from 8% up to 32% due to a different driver's "anxiety".

2.5.2 EVSE Occupation

The figure 2.9 shows two histograms representing how much the charging stations are occupied in percentage (over time). It compares different results according to the number of electrical vehicles involved in the simulation and it shows in parallel how many requests were not satisfied with reservation enabled (in red) and without reservation enabled (in green). A request is considered not satisfied when either

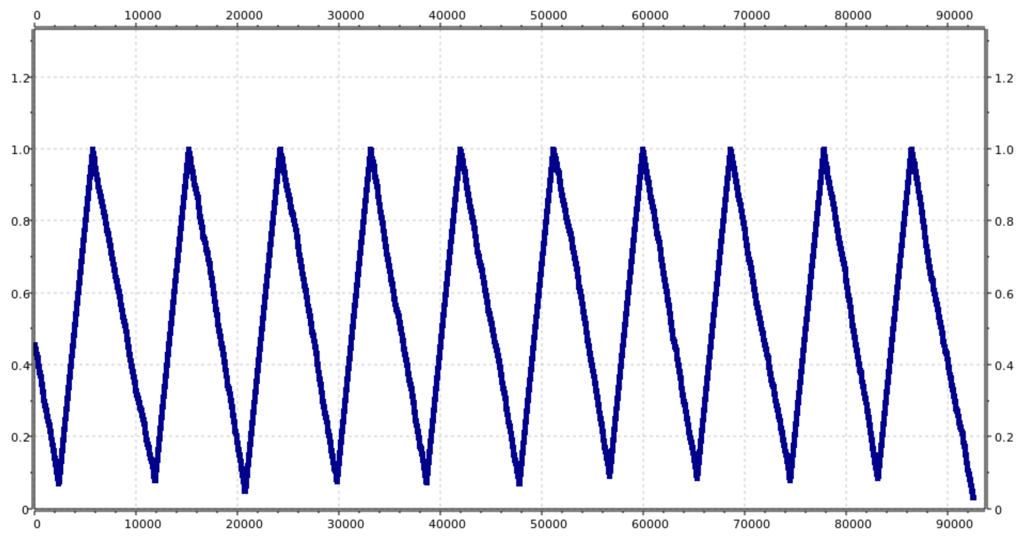


Figure 2.7: State of charge of a vehicle in a 28-hour simulation.

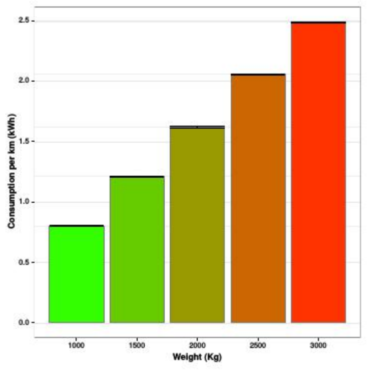


Figure 2.8: Vehicle consumption in relation with its weight [24].

the driver drives away from a charging station due to the length of the waiting line, or the selected option is rejected (in case of reservation enabled), or even it runs out of battery before managing to recharge it. It is obvious how a reservation mechanism is necessary for such an infrastructure to avoid huge waiting lines.

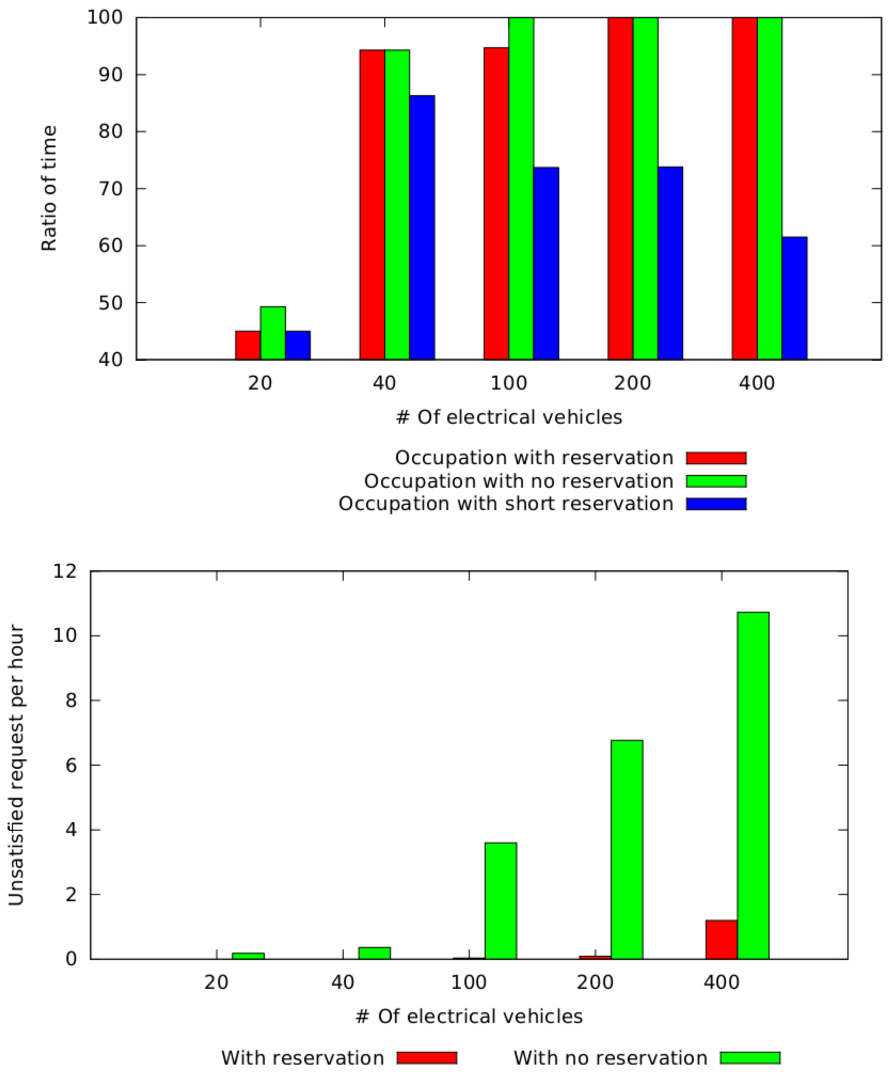


Figure 2.9: EVSE overall occupation with and without reservation (above) and number of unsatisfied requests (too long queues, options rejected...) [24].

Chapter 3

The Arrowhead Paradigm

3.1 Arrowhead Core Entities

In this section we will focus on the Arrowhead main entities, which are the gravitational center for all the interactions among any number of Arrowhead systems. As stated in section 1.1.1, any SOA-based infrastructure (called System -of-Systems) needs service consumers, service providers and one or more third-party service brokers. Those brokers are services themselves and are called Arrowhead Core Services (ACS) and are supposed to be hosted in the same machine that runs the service broker daemon (or the same subnet). The ACS are highly responsible for the interaction among any other system in the Arrowhead framework. So far, precisely at the beginning of Arrowhead Generation II [25], the ACS are unique and centralized, but there are future expectations on enabling their scalability, both in horizontal direction (more ACS of the same type in a cloud or a cluster) and in vertical direction (hierarchy of different types of ACS for different domains). As stated in paragraph 1.2.2, the ACS should implement the Information Infrastructure, the System Management and the Information Assurance. This stage has already been reached during the Arrowhead Generation I [26] and plenty of documentation is provided about those systems. Here we report a brief description of the ACS so far implemented.

3.1.1 Service Registry

The Service Registry is the main service present in the ACS. Its purpose is acting as the service broker itself and implements the Information Infrastructure. It keeps track in a table which services are registered and published to make them reachable for every service consumer whatsoever. The connection to the Service Registry and the operations of service discovery are modeled using the standard RFC 6763: DNS-SD [52], a protocol based on DNS lookups. Each record published by a service provider on the Service Registry must be composed by the following fields:

- Service Name, an unique name to identify the service.
- Service Type, a DNS-SD standard notation for the service type: `_servicetype._transportprotocol`. It is a good practice to split the above mentioned `_servicetype` in `_servicetype_applicationprotocol`. All in all, the Service Type cannot be more of 14 characters long; an example could be `_temp_rest-ws._tcp` for a service providing temperature data, using a REST Web service and TCP as a transport protocol.
- Service Instance, an unique name which is compound by `ServiceName.ServiceType.BrowsingDomain` and is officially designed as the service unique identifier. An example could be `temp001._temp_rest-ws._tcp.srv.arrowhead.eu`.
- The Service Endpoint, the complete host identifier in which the system hosting the service is running, followed by port and path by which the service is reachable. As is a DNS-based interaction, the host should consist of a hostname, not a plain IP address.
- The Service Metadata, a chunk of properties mapped as "[name]=[value]" which denote additional information about the service (such as the version).

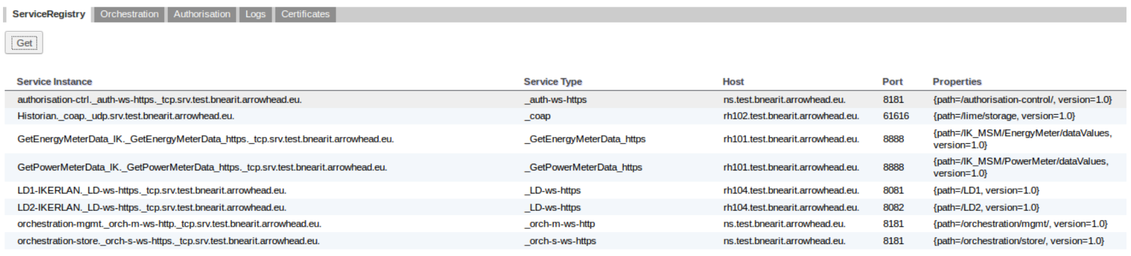
Any system willing to consume one of the services published in the Service Registry performs first of all the operation of **Discovery**, which consists in reading the records stored in the registry to find the service needed. Once such a service is found, the consumer stores the whole service record, in particular the endpoint in which the service is running. The action of consuming a service is then a simple client-server interaction in which the consumer connects directly to the endpoint previously discovered.

At the same time, any provider may perform the actions of **Publish** and **Unpublish** against the Service Registry. These operation respectively add and delete a record to the Service Registry to make the Service provider reachable from any Arrowhead-compliant consumer.

In terms of design, the Service Registry system consists of the following service interfaces [21]:

- **Service Discovery (provided)**: it is the service consumed by all the Arrowhead systems willing to perform Publish, Unpublish and Discovery operations (basically any Arrowhead system must perform at least one of these operations).

Figure 3.1 shows the graphical web interface, called Management Tool (which itself consumes the Service Discovery DNS-SD), showing all the service records published at the moment.



Service Instance	Service Type	Host	Port	Properties
authorisation-ctrl._auth-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_auth-ws-https	ns.test.bnearit.arrowhead.eu.	8181	{path=/authorisation-control/, version=1.0}
Historian._coap._udp.srv.test.bnearit.arrowhead.eu.	_coap	rh102.test.bnearit.arrowhead.eu.	61616	{path=/lime/storage, version=1.0}
GetEnergyMeterData_IK._GetEnergyMeterData_https._tcp.srv.test.bnearit.arrowhead.eu.	_GetEnergyMeterData_https	rh101.test.bnearit.arrowhead.eu.	8888	{path=/IK_MSM/EnergyMeter/dataValues, version=1.0}
GetPowerMeterData_IK._GetPowerMeterData_https._tcp.srv.test.bnearit.arrowhead.eu.	_GetPowerMeterData_https	rh101.test.bnearit.arrowhead.eu.	8888	{path=/IK_MSM/PowerMeter/dataValues, version=1.0}
LD1-IKERLAN._LD-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_LD-ws-https	rh104.test.bnearit.arrowhead.eu.	8081	{path=/LD1, version=1.0}
LD2-IKERLAN._LD-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_LD-ws-https	rh104.test.bnearit.arrowhead.eu.	8082	{path=/LD2, version=1.0}
orchestration-mgmt._orch-m-ws-http._tcp.srv.test.bnearit.arrowhead.eu.	_orch-m-ws-http	ns.test.bnearit.arrowhead.eu.	8181	{path=/orchestration/mgmt/, version=1.0}
orchestration-store._orch-s-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_orch-s-ws-https	ns.test.bnearit.arrowhead.eu.	8181	{path=/orchestration/store/, version=1.0}

Figure 3.1: Management Tool, portion of the Service Registry section.

3.1.2 Authorisation

The Authorisation system is the system implementing Information Assurance capabilities within the ACS. In particular, it keeps in memory a chunk of records which state who is allowed to consume certain services. Furthermore, a service provider may or may not require authentication control, as it may be a global provider not requiring any kind of authentication. Anyhow, the possibility of interaction with the Authorisation services is mandatory for any Arrowhead-compliant system whatsoever. As a common use case, a service provider, after receiving a service request by a consumer, it checks by connecting to the Authorisation system whether the consumer is effectively allowed to use the service before providing it.

The records stored in the Authorisation system are based on X.509 certificates (standard RFC2459) [53]. Any consumer of a secure service must therefore have a trusted certificate and, furthermore, obey to the authorisation records. An authorisation record is composed by the following fields:

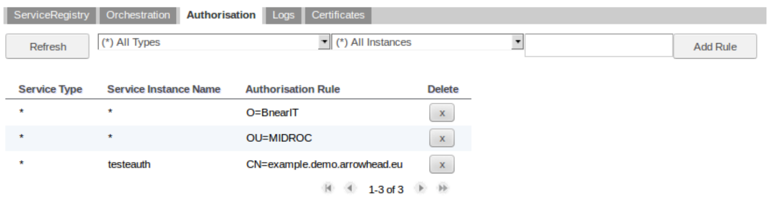
- Service Type, which is the DNS-SD Service Type of the service reachable (i.e. service that can be consumed) by this rule. A * means all types.
- Service Instance, which is the DNS-SD Service instance of the service reachable by this rule. A * means all instances.
- Authorisation Rule, which is a set of association "[attribute]=[value]" where the attribute is one of the common RDNs of a X.509 certificate: CN (Common Name), OU (Organizational Unit), O (Organization), L (Locality), S (State), C (Country). An example may be "O=BnearIT;C=S", which means

that only who has a certificate validated by the company BnearIT in Sweden may consume the service specified by the other two fields.

In terms of design, the Authorisation system consists of the following service interfaces [22]:

- **Authorisation Control (provided)**: it is the service which provides read access to the authorisation rules. Any secure service provider needs to access it.
- **Authorisation Management (provided)**: it is the service which provides write access to the authorisation rules in order to insert, modify or delete rules.
- **Service Discovery (consumed)**: the Authorisation system needs to consume this service in order to be published.

Figure 3.2 shows the Management Tool, which consumes both Authorisation Control and Authorisation Management, showing all the authorisation rules valid at the moment.



Service Type	Service Instance Name	Authorisation Rule	Delete
*	*	O=BnearIT	x
*	*	OU=MIDROC	x
*	testeauth	CN=example.demo.arrowhead.eu	x

1-3 of 3

Figure 3.2: Management Tool, portion of the Authorisation section.

3.1.3 Orchestration

The Orchestration system is the system implementing System Management capabilities within the ACS. In particular, it keeps in memory a chunk of records called orchestration configurations. Those configuration are used when multiple instances of a certain service are published, to redirect dynamically a consumer to one of those. A consumer which needs a service of a particular type may not be interested in getting it manually but rather relying on which instance the Orchestration system redirects the consumer to.

The interaction with the Orchestration system is currently avoidable (i.e. a consumer may just retrieve the service roughly from the Service Registry), but interoperability with the Orchestration store is one Arrowhead requirement and thus it must be implemented by each Arrowhead system.

The most common use case is given by a service running in several instances. The service provider(s) are run by a company which own a third entity, an Arrowhead "monitor" capable of interaction with the Orchestration management. A consumer is connecting to one of the services and thus the Orchestration store stores a new record which associates the consumer with the service instance. If the provider producing such instance faults, the "monitor" entity may change the orchestration record associating a second instance to the consumer. The consumer itself must periodically monitor the orchestration store and, when perceives a new association, must switch the service instance to consume.

In terms of design, the Orchestration system consists of the following service interfaces [23]:

- **Orchestration Store (provided)**: it is the service which provides read access to the orchestration records.
- **Orchestration Management (provided)**: it is the service which provides write access to the orchestration records.
- **Authorisation Control (consumed)**: as the Orchestration system is meant to be secure, it must check whether a system is authorized to access it.
- **Service Discovery (consumed)**: the Orchestration system needs to consume this service in order to be published.

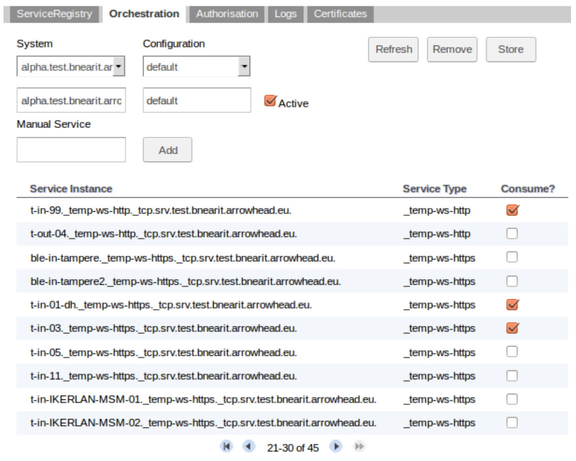
Figure 3.3 shows the Management Tool, which consumes both Orchestration Store and Orchestration Management, showing all the orchestration configurations valid at the moment.

3.2 Glossary of the Terms

As the Arrowhead project is shared among several partners, a common and clear glossary to define all the single entities is required [27].

3.2.1 Concepts

This subsection states clearly what we mean with a certain term in a unique way, to avoid ambiguities. To have a broader explanation [27] is needed.



System

alpha.test.bnearit.ar

Configuration

default

alpha.test.bnearit.arrc

default

✓ Active

Manual Service

Service Instance	Service Type	Consume?
t-in-99._temp-ws-http._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-http	✓
t-out-04._temp-ws-http._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-http	☐
ble-in-tampere._temp-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-https	☐
ble-in-tampere2._temp-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-https	☐
t-in-01-dh._temp-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-https	✓
t-in-03._temp-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-https	✓
t-in-05._temp-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-https	☐
t-in-11._temp-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-https	☐
t-in-IKERLAN-MSM-01._temp-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-https	☐
t-in-IKERLAN-MSM-02._temp-ws-https._tcp.srv.test.bnearit.arrowhead.eu.	_temp-ws-https	☐

21-30 of 45

Figure 3.3: Management Tool, portion of the Orchestration section.

- **System**: a System is what is providing or consuming services (it must be a Service Provider or a Service Consumer or both) and it is normally executing a program on a hardware (it may be a personal computer or a small sensor).

- **System-of-Systems**: a System-of-Systems is a set of Systems which are communicating with each other strictly using the Arrowhead Framework. It includes thus the ACS.

- **Service**: the definition of Service has been given in section 1.1.1. In the Arrowhead Framework it has to be defined using a precise documentation profile, it may be compound and is produced by a Service Provider, consumed by a Service Consumer and follows all the paradigms in a common SOA.

- **Service Interface**: it is the output or input interface for a service, described independently from the technology using a standard format. In each diagram a service interface is denoted with the "lollipop notation", thus an inbound service interface (the interface provided by the Service Provider) is represented by a full lollipop, while an outbound service interface (the interface instantiated remotely by a Service Consumer) is represented by an empty half lollipop.

- **Service Provider**: it is a system which produces services and makes them available to a set of Service Consumers through publishing.

- **Service Consumer**: it is a system which discovers and consumes services made accessible by Service Providers.

- **Black Box**: it is a description of a system in terms of its outer interfaces, interface descriptions and communication protocols without any knowledge about the internal implementation.

- **White Box**: it is the complete description of a system; it may be seen as the implementation of a Black Box.

- **Legacy System**: it is a system which is not compliant with the Arrowhead Framework.

- **Legacy Adapter**: it is a piece of software which allows a legacy system to be Arrowhead-compliant.

- **Maturity Level**: it is, on a scale from 0 to 5, a measurement of how a system is compliant with the Arrowhead Framework. In particular, they are shown in figure 3.4 and are: Legacy system, Thin Client, Legacy Interface, Arrowhead/Legacy Interface, Arrowhead Interface and Arrowhead Compliance.

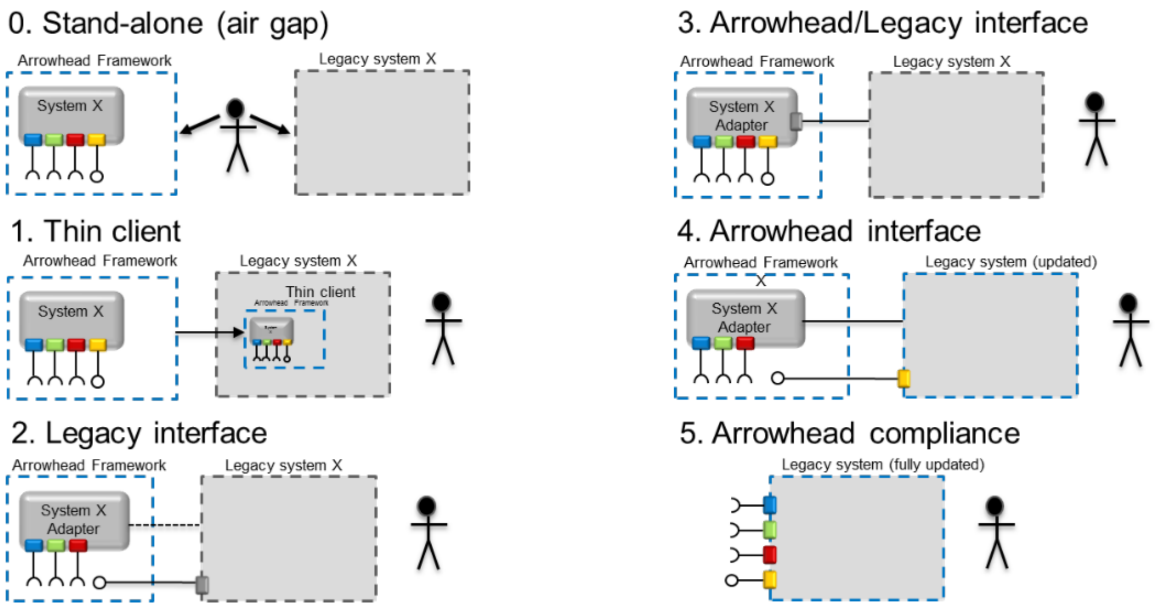


Figure 3.4: Graphical representation of the maturity levels [26].

3.2.2 Documentation Elements

This section states clearly which documentation should be produced to define any newly deployed Service, System or System-of-Systems. To have a broader explanation [27] is needed.

- **System-of-Systems Description (SoSD)**: is a high-level view of a System-of-Systems seen as a Black Box, thus describes functionalities and architecture. It refers to a number of SysDs.
- **System-of-Systems Design Description (SoSDD)**: is a high-level view of a System-of-Systems seen as a White Box, thus describes the implementation of the functionalities. It refers to a number of SysDDs.
- **System Description (SysD)**: is a description of a System as a Black Box, thus describes its service interfaces and purposes. It refers to a number of IDDs and SDs.
- **System Design Description (SysDD)**: is a description of a System as a White Box, thus describes how the system was designed and implemented.
- **Service Description (SD)**: is an abstract description of a Service, stating its interfaces, purposes and information type.
- **Interface Design Description (IDD)**: is a specific description of a Service. It can be seen as the instantiation of an SD with a specific technology.
- **Semantic Profile (SP)**: it is the description of the data format exchanged within the scope of a certain service (i.e. the encoding). It is referred to by an IDD.
- **Communication Profile (SP)**: it is the description of the different protocols used by a service in its connection. It comprehends the transfer protocol, the security mechanism and the data format (specified also in the SP). It is referred to by an IDD.

3.3 Reason in an "Arrowhead Way"

Whenever an agent, from a single developer to a whole company, wants to create an Arrowhead-compliant application, which can act as both a Service Provider and a Service Consumer, should start planning smartly how to represent data and how to be interoperable with other agents. Enabling de facto standardization is very important within the scope of such a big project and often partners, if the dissemination is not done properly, may run into small clusters among which the data exchanged is not in the same format and does not respect the same paradigm. To avoid this situation an "Interoperability Matrix" has recently been presented in the last Arrowhead meetings [28]. The matrix shows currently that partners are roughly not using the same encoding for service types, thus the next step for the project is fostering new awareness of the global picture.

Designing a new service producer requires that the developer must encode the information that is meant to be provided in a service oriented paradigm. This means:

- Produce each information in terms of service.
- Encapsulate all the informations a consumer must know in order to interact with the service in a clear and standardized document (the IDD).
- Look up (using the interoperability matrix) who else among the partners used the same type of information and try to use the same encoding (or agree on a standard).
- Ponder which granularity must be assigned to the service provided; in some cases may be useful splitting the service in several "service atoms" which can be aggregated to produce compound functionalities. A too coarse granularity may lead to some missing functions and may lack loose coupling and customizability. On the other hand, a too fine granularity may lead to flooding the Service Registry with several pointless records that would unlikely be singularly consumed and would require more effort from the consumer's developer in order to aggregate them.

Designing a new Service Consumer, on the other hand, means being aware that the interactions among systems are all mediated by the Arrowhead Framework and, to achieve interoperability among the parts, a consumer must develop an interface following the standards and the design stated in the producer's IDD and SD.

Chapter 4

Arrowhead REST User-Friendly Java API

4.1 Structure of the API

One of my personal main contribution to the project consist of user-friendly API for Arrowhead applications written in Java and compliant with the REST architecture and underlying protocols. It is usable even with other communication protocols, but, in those cases, the API offers less support and forces the developer to implement some parts of the underlying layers.

4.1.1 Motivations and Requirements

Whenever interacting with a common framework needs standardization it is necessary to guide a developer through a certain amount of steps both to simplify his or her implementation against that framework and to avoid misuse of some functions. The company BnearIT, as a part of WP8, developed two packages which encapsulate the functions to interact with the ACS: `core-utils-1.4`, used to interact with basic publish, unpublish and discovery functions, and `dnssdjava-1.0-beta7`, used to interact with the ACS through the DNS-SD protocol. Those libraries are not open-source (neither among the Arrowhead partners).

Hence, building the APIs discussed in this chapter has been found to be necessary due to several reasons:

- There is either little or sparse documentation on how to build up an Arrowhead-compliant application; it is really tricky especially for those who never attended any of the workshops organized.

- The code of the above mentioned libraries is not visible, thus it is important to wrap them with good APIs in order to help the developer in understanding what is effectively happening at lower layers.
- Provide an unique working interface to all the Arrowhead partners, which should be as less protocol-dependent as possible.
- Organize the code into a layered paradigm, to let the end developer dealing with only the highest layers.
- Make possible for the developer to get a "hello world" application with really few lines of code.

According to these motivations and their goals, the APIs have been organized in three different conceptual layers, which allow the developer to interact the less necessary with them.

1. The first layer is supposed to be common to all the Arrowhead applications. It has built-in support for REST applications using Jetty [54] as an HTTP server together with Jersey as an API for RESTful Web services, but it can be used with every protocol whatsoever.
2. The second layer is not released as usable, because it is supposed to be common to all the Arrowhead applications within the same domain (i.e. Arrowhead Applications which use the same language, the same protocol, the same data format and nearly the same producing/consuming processes). It is released as a "hello world" example and it is REST-compliant.
3. The third layer is the end application, which is normally supposed to consist of few line of Java code.

The APIs are released under GPL license among the Arrowhead partners.

4.1.2 Layer 1 (Common to all the Arrowhead Applications)

The first layer of the API, together with the "hello world" sample second layer, is retrievable from the Arrowhead SVN repository at the address `https://forge.soa4d.org/svn/arrowhead/WP9/Task9.3/Working/UniBO/`. It depends on several libraries (included in the project as well), here we report briefly some of them to give an idea:

- Java REST Webservice (java-ws-rs).
- Jetty Web server.

- Jersey REST API.
- Log4J, a Java-based logging utility.
- BnearIT Core Utils.
- BnearIT DNS-SD utilities.

The main concept on which the whole API is based is the design pattern "Abstract Factory". Using this pattern, a single object, called factory, is able to produce different objects given different parameters in input. In this case each service is published and managed (and run in some cases) by a single Service Producer, which is an object created by an abstract factory. Different types of services are managed by different types of Services Producers, for this reason an interface called AppServiceProducer has been implemented as a basis for each different Service Producer. Specularly, each service is consumed by a different object, called Service Consumer, which implements in turn the common interface AppServiceConsumer and is generated by another abstract factory. Each time a service needs to be published a producer object will be instantiated to manage the whole service's life cycle. Each time a resource needs to be consumed a new consumer object will be instantiated to manage he whole life cycle of that particular consumption. To avoid an excessive creation and deletion of object, a recycle mechanism has been introduced by which objects are never deleted, but rather saved in a pool and ready to be reused whether a new object needs to be instantiated.

This solution has been adopted to avoid that the end developer interacts uncontrolled directly with the service or the endpoint, therefore the access to those structures would be guided by intermediary objects. Logically, each domain requires to instantiate objects from different producer classes as well as consumer classes. The layer 1 API provides only a general Service Producer class and no Service Consumer class, while the interfaces are provided. Layer 2 is where the Service Producer and Service Consumer classes are implemented for each specific domain.

The main entities in these APIs are two important objects, which act as a bottleneck for almost any Arrowhead-related function call. The first one is called *ArrowheadController* and has to be instantiated before anything else. By calling this object, with the name of the system as an input for the constructor, the developer is supposed to have in the project main directory the following files:

- A property file which must have the same name as the system followed by a `.properties`. This file collects almost all the parameters necessary to instantiate the interaction with the ACS.

- A JKS file, a keystore (pointed by the properties file) which is generated by the Java Keytool from a set of X.509 certificates. It is used to perform secure connections to the services and the certificates have to be released by trustworthy authorities according to the ones registered in the ACS.
- A Transaction Signature file, called TSIG (pointed by the properties file as well), which is needed by any provider to being able to publish a service on the ACS. It is a plain text file containing both the key name and the full key and it has to match the couple registered on the configurations of the DNS server hosting the ACS.

If these requirements are satisfied, the ArrowheadController parses the property file, keeps in memory the pointers to the ACS (IP address and domain name) and to the local hostname, stores the necessary security files and configures a LOG file (which, by default, has the same name as the system). Subsequently, the developer has to specify which types of data need to be produced or consumed and which type of Service Producer or Service Consumer class is in charge for that particular type. By this call, the ArrowheadController keeps a list of Producer and Consumer factories, each of them associated with a set of service types and a Service Producer or Service Consumer class type.

The other main object is the *ArrowheadSystem*, which needs to be instantiated after the above mentioned operations are performed. During the instantiation it connects to the ACS using the parameters specified in the properties file and provides the developer with a large set of function calls. Using those functions, the developer will be able to:

- Create a new producer object: by specifying the service type, the system checks which of the factories is in charge for that particular type and calls the factory's creation function.
- Create a new consumer object: by specifying the service type, the system checks which of the factories is in charge for that particular type and calls the factory's creation function, or gets one of the unused Service Consumer objects from the pool.
- Perform a service discovery, which can retrieve all the published services or get them filtered by name or type.
- Getting any service property: name, type, endpoint, metadata and instance.
- Getting a raw resource endpoint, used both for highly customized interactions as well as legacy client integrations.

- Getting or destroying any Service Producer or Service Consumer object by name.
- Erasing a service record from the Service Registry given the name.

Furthermore, a Service Producer object is used for publishing the handled service, unpublishing it, running and stopping the server hosting the service (if the system is not used just as a publisher). A Service Consumer object, in turn, is instantiated after a discovery against a single service and is used for consuming that service using either no parameter, parameters encoded in a string or parameters encoded in a Form object.

4.1.3 Layer 2 (Common to the Arrowhead Applications within a Specific Domain)

The second layer of the API is dedicated to the domain-specific developing area. This means that the layer 2 API provided (available in the repository together with the layer 1 APIs) are just a "hello world" example, useful as a tutorial on how to actually develop it. A fully compliant layer 2 API has to implement the following classes:

- A certain number of Service Producer classes, depending on how many service types the system is meant to produce. Those classes have to implement the interface AppServiceProducer, or, for better compliance and transparency, extend the class GeneralPublisher (provided in layer 1 and implementing AppServiceProducer plus other useful functions). Furthermore, If the Service Producer is meant to be a Java RESTful Web service, may be useful to implement the producer's "resource" field (the only difference with the general Service Producer class, which instantiates it to null), which has to be instantiated and implemented (in a separate class) according to the Java WS RS paradigm. In any other case it is possible to override the *start()* and *stop()* functions to implement the server's behavior.
- A certain number of Service Consumer classes, depending on how many service types the system is meant to consume. Those classes have to implement the interface AppServiceConsumer, or, for better compliance and transparency, extend the abstract class ArrowheadServiceConsumer (provided in layer 1 and implementing AppServiceConsumer plus other useful functions).
- The above mentioned resource class (a detailed example can be found in the sample provided).

- Optionally, a certain number of customized factories, one for each service type, if the implementation needed is different from the default implementation present in layer 1.

Once these classes are implemented, the developer may start to write the layer 3, which is the application itself (explained in the following section).

4.2 Using the API

This section is about how an end developer should interact with the APIs, thus how to implement the layer 3 application.

All those different use cases require a well formed properties file to get the informations from. It should have the typical Java Properties format and the following fields are mandatory:

- **core.server**: the IP address of the ACS (e.g. `10.200.0.10`).
- **core.domain**: the Domain Name of the ACS (e.g. `test.bnearit.arrowhead.eu`).
- **core.hostname**: the valid hostname of the system (in case the publisher and the provider are running in different locations, the hostname of the provider should be given) (e.g. `rh105.test.bnearit.arrowhead.eu`).
- **core.tsig**: the name of the TSIG file (e.g. tsig).
- **truststore.file**: the path to the truststore file.
- **truststore.password**: the password for the truststore file.
- **keystore.file**: the path to the keystore file.
- **keystore.password**: the password for the keystore file.
- **authorisation.url**: the URL to the authorisation control service in case it was not found on the ACS (e.g. `https://10.200.0.10:8181/authorisation-control`)
- **orchestration.url**: the URL to the orchestration store service in case it was not found on the ACS (e.g. `https://10.200.0.10:8181/orchestration/store`)
- **orchestration.monitor.interval**: amount of time passing between each orchestration check in seconds.

- **service.consume.support**: service types that the system is allowed to consume, separated by a pipe (e.g. `_hello-ws-http._tcp|_hello-ws-https._tcp`).

4.2.1 Developing a Publisher for a Legacy Provider

When a company or a team needs to integrate a legacy system with the Arrowhead Framework, in order to make it reachable without the need for complete compliance, it is required an adapter, which, in this case, is a small program using the proposed APIs. The programs gets the data related to the service(s) to publish and simply connects to the ACS and performs a publish operation. For this purpose, as can be seen in figure 4.1, only layer 1 is needed.

First of all instantiating the ArrowheadController is needed.

```
static ArrowheadController arrowheadController = new
    ArrowheadController(systemName);
```

Since this program does not require any special Service Producer object (it uses the default one provided in L1), we need to perform the connection to the ACS by instantiating the ArrowheadSystem.

```
static ArrowheadSystem arrowheadSystem = new ArrowheadSystem();
```

After done this, recall that our service is running on the host denoted by the "core.hostname" property, we need to instantiate a Service Producer object and publish the service. The function *createPublisher* tries to create a Service Producer object using the factories available and, if no factory is available for that type, it creates a factory for the default Producer and associates it with the service type passed.

```
AppServiceProducer producer = arrowheadSystem.createPublisher(
           "hello-world",        // service name
           "_hello-ws-http._tcp", // service type
           "80|root/path",       // port | path
           "version=1.0");        // metadata, currently not in use
producer.publish();
```

The published service is supposed to run on `hostname:80/root/path`. When the record is no longer needed it can be unpublished:

```
producer.unpublish();
```

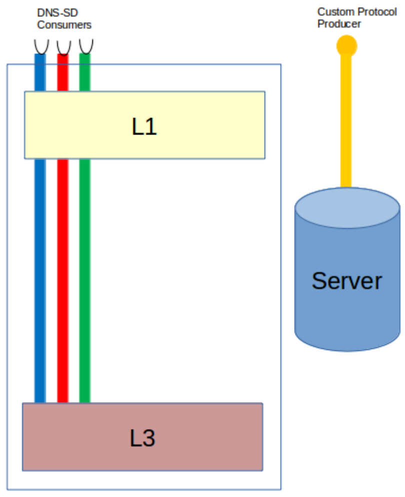


Figure 4.1: Decoupled Service Provider: a publisher and an external server.

Furthermore, it is possible to erase a service from the Service Registry if the name is known:

```
arrowheadSystem.eraseService(serviceName);
```

4.2.2 Developing a Discoverer for a Legacy Consumer

As stated above, there may be a need to integrate a legacy system in the Arrowhead Framework. The integration may happen as well for a client, which needs to act as a Service Consumer. This subsection is a short tutorial on how to build an adapter for a client, which merely needs to fetch the complete URL of a determined resource.

After instantiating the ArrowheadController, the Consumer does not have a default Service Consumer class, but this approach may completely avoid to use the factories directly approaching to the discovery operation. Indeed, as stated in figure 4.2, the layer 2 APIs are again not needed.

```
static ArrowheadController arrowheadController = new
    ArrowheadController(systemName);
static ArrowheadSystem arrowheadSystem = new ArrowheadSystem();
```

After the initialization step, we need to discover the service to consume. This operation may be performed in one of the following three ways:

```
ServiceIdentity identity = arrowheadSystem.getServiceByName(serviceName);
List<ServiceIdentity> identities =
    arrowheadSystem.getServicesByType(serviceType);
List<ServiceIdentity> identities = arrowheadSystem.getAllServices();
```

Clearly, if getting a list of services is decided, we need to scroll among them to find the designed one. We may get informations about a service identity though the following operations:

```
String instance = identity.getId();
String type = identity.getType();
ServiceEndpoint endpoint = arrowheadSystem.serviceGetEndpoint(identity);
ServiceMetadata metadata = arrowheadSystem.serviceGetMetadata(identity);
```

Once we find the designed service we may get the URL of a certain resource we are interested in consuming. The URL may be sent to our legacy client application afterwards.

```
URL url = arrowheadSystem.serviceGetCompleteUrlForResource(identity,
    "someresource.php");
```

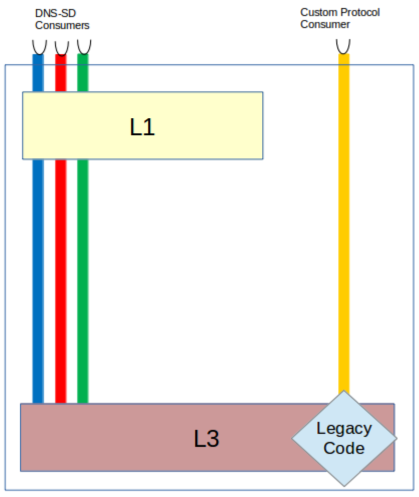


Figure 4.2: Simplified Service Consumer: a discoverer and a legacy code.

4.2.3 Developing a Compact Service Provider

This subsection is about developing a Service Provider from scratch. Clearly, we need to be as compliant as possible, thus the Service Provider must have the maximum maturity level. Before starting to develop it we need to have a layer 2 API which implements the interface AppServiceProducer. For full compliance the class GeneralPublisher may be extended and an additional class must be developed as a REST Web service resource, as stated before. Alternatively the AppServiceProducer may be implemented completely overriding the methods for starting and stopping the server to achieve full customizability.

For the following example we assume that our Service Producer class is called *HelloProducerREST_ WS*. After instantiating the ArrowheadController, we need to generate a factory for the service type(s) that we mean to produce and associate it with the Service Producer class in charge for producing them.

```
static ArrowheadController arrowheadController = new
    ArrowheadController(systemName);

arrowheadController.addNewProducerBinding(
   Arrays.asList("_hello-ws-http._tcp", "_hello-ws-https._tcp"),
   HelloProducerREST_WS.class);

static ArrowheadSystem arrowheadSystem = new ArrowheadSystem();
```

After this initial settings we can now create our Service Producer object, publish it and start it (that is, start the server producing the service).

```
AppServiceProducer producer = arrowheadSystem.createProducer(
            "hello-world",        // service name
            "_hello-ws-http._tcp", // service type
            "80|root/path",       // port | path
            "version=1.0");         // metadata, currently not in use
producer.start();
producer.publish();
```

This function creates indeed a *HelloProducerREST_ WS* instance, because we declared that this class is in charge for the service type selected. It is slightly different from the function *createPublisher*, as in this present case, if no proper factory is found, an error is returned. When the service is no longer needed it is simply unpublished and stopped:

```
producer.unpublish();
producer.stop();
```

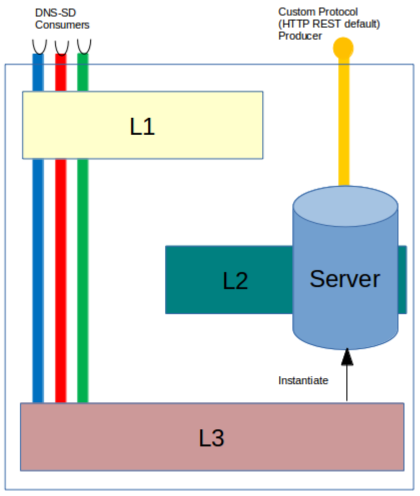


Figure 4.3: Compact Service Provider.

4.2.4 Developing a Compact Service Consumer

This subsection is about developing a Service Consumer from scratch. It addresses full Arrowhead-compliance and needs clearly a layer 2 implementation. In particular, it needs to implement the Service Consumer class either extending the abstract class ArrowheadServiceConsumer or directly implementing the interface AppServiceConsumer. In the first case, the class is more compliant, even if REST architecture is not required in this case, and just needs to implement the following methods:

```
//consume a resource with no parameters
abstract public String consumeResource(String resource);
//consume a resource with parameters encoded in a string
abstract public String consumeResource(String resource, String params);
//consume resource with parameters encoded in a form
abstract public String consumeResource(String resource, Form form);
```

Once the Service Consumer class is ready (in our example it will be called *HelloConsumerREST_WS*) the layer 3 can be implemented. As above, after instantiating the ArrowheadController, we need to specify which types our application will consume and which Service Consumer object will handle that operation.

```
static ArrowheadController arrowheadController = new
    ArrowheadController(systemName);

arrowheadController.addNewConsumerBinding(
    Arrays.asList("_hello-ws-http._tcp", "_hello-ws-https._tcp"),
    HelloConsumerREST_WS.class);

static ArrowheadSystem arrowheadSystem = new ArrowheadSystem();
```

The discovery process is the same as the one presented in 4.2.2. After we got the service identity we aim to consume, we need to create a Service Consumer object and lock it to avoid that some other concurrent process uses it (i.e. to mark it as “in use”).

```
AppServiceConsumer consumer =
    arrowheadSystem.createConsumer("hello-world");
consumer.lock();
```

Indeed, in this case the consumer returned will be an instance of *HelloConsumerREST_WS* due to the association specified at the beginning. Once the consumer is correctly pointing to the resource we need to physically consume it (as implemented in our class).

```
String result = consumer.consumeResource("someresource.php");
```

When the consuming process is over we can release the Service Consumer object.

```
consumer.unlock();
```

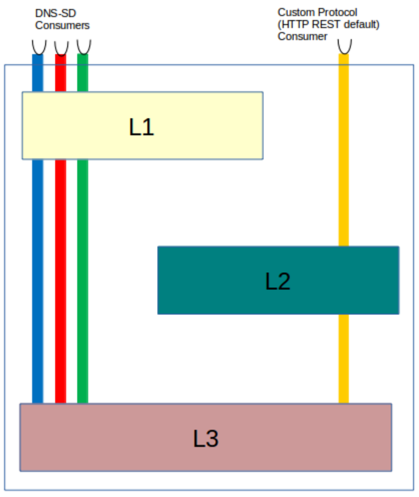


Figure 4.4: Compact Service Consumer.

Chapter 5

Arrowhead Service-Oriented Integration

In this chapter it is pointed out how the integration of the simulator within the Arrowhead Framework took place. We aim to describe both the real scenario and the simulated scenario and, subsequently, how the Arrowhead adapters work.

This system, developed in the scope of WP9, Task 9.3, has been carried out following the directions about how the same scenario would be organized in the real world. The following subsections are about how WP3 designed the scenario, the use cases and the interactions and how the present project simulated them in parallel.

5.1 The Real World Arrowhead E-Mobility Scenario

The work planning performed by WP3 is nowadays ongoing, thus the references pointed out in this report are referring only to the current state of the art. The main scenario is considered as the same scenario presented in the E-Mobility Simulator: an urban scenario with a certain number of electrical vehicles and charging stations, a booking service and, mainly, users, which can interact with their vehicles (and other services as well) using a mobile application in their mobile devices.

5.1.1 Use Cases

The scenario is presented under various views, depending on the use cases we are interested in. The following use cases have been defined by WP3:

1. The most important use case is given by the reservation facilities. An user must be able to interface with a booking system and perform a reservation submitting a preferred time and place and shall be able to choose among the offered possibilities which are given according to which charging station is not busy, reserved or faulted.

2. The charging stations must be able to consume an external weather forecasting service to manage their solar panels as an alternative source of energy.

3. The private charging stations must be able to consumer external pricing services such as FlexOffer.

4. An user must be able to monitor the recharging process parameters while his or her vehicle is recharging at a public charging station.

5. The charging stations must provide information about their statuses and electrical parameters (those informations may be used, for example, by the reservation infrastructure).

6. A charging station must be able to verify if an user is correctly reserved to start the recharging process.

5.1.2 Systems and Services

The above mentioned use cases leaded to the development of two main system concepts: the *Booking System* and the *Management System.*

The Booking System is an Arrowhead-compliant system which aims to receive charge requests from the users and offers a set of options based on the input parameters. This is offered by its *Booking Service*, an unique service provided to the users, thus it is a Service Provider. Furthermore, it needs to fetch continuously informations about the charging stations as well, so it acts as a Service Consumer too, since the charging stations are offering the *Monitoring Service.* The Booking Service is thus offering the following functions:

- Insert a charge request to the system.
- Confirm a charge option to the system.
- Verify a reservation.
- Retire a reservation.

The Management System is an Arrowhead-compliant system which aims to instantiate a single *Monitoring Service* for each charging station. This service provides a set of informations about the charging station itself, especially regarding status, electrical parameters and price. These information are useful to the Booking System in the first place, and to any actor interested in being updated. Furthermore, the Management System needs to consume the Booking System, as it may need to verify if a user is correctly reserved or not, plus it consumes the external services FlexOffer and Weather Forecasts, thus it is both a Service Provider and a Service Consumer. The Monitoring Service is thus offering the following functions:

- Get status data about the charging station.
- Get the current charging process parameters.
- Get the position of the charging station.

5.1.3 Structure

As a big picture design, we can distinguish among three different main scenarios: the EVSE monitoring scenario, the booking scenario and the recharge scenario. In this section all these scenarios are presented in one unique big picture, to show in sufficient detail how the entities interact. The System-of-Systems is given by the union of all the elements and the interactions involved, shown in figure 5.1.

The common entities to all the scenarios are:

- The ACS.
- The set of all the real charging stations and vehicles.
- A cloud, which is responsible for the storage of every single information about the public charging stations, connected to it by a custom framework called KURA. It hosts also the Management Systems.
- A separate Booking System, which is composed by a central storage system, probably a SIB, a logical unit (which processes all the input from the extern) and an Arrowhead adapter which exposes the Booking Service.

EVSE Monitoring Scenario

This scenario involves the Arrowhead Service Producer referred to as Management System, already defined. The scenario shows a custom arrowhead consumer, which may be represented by a real mobile application, consuming the Monitoring

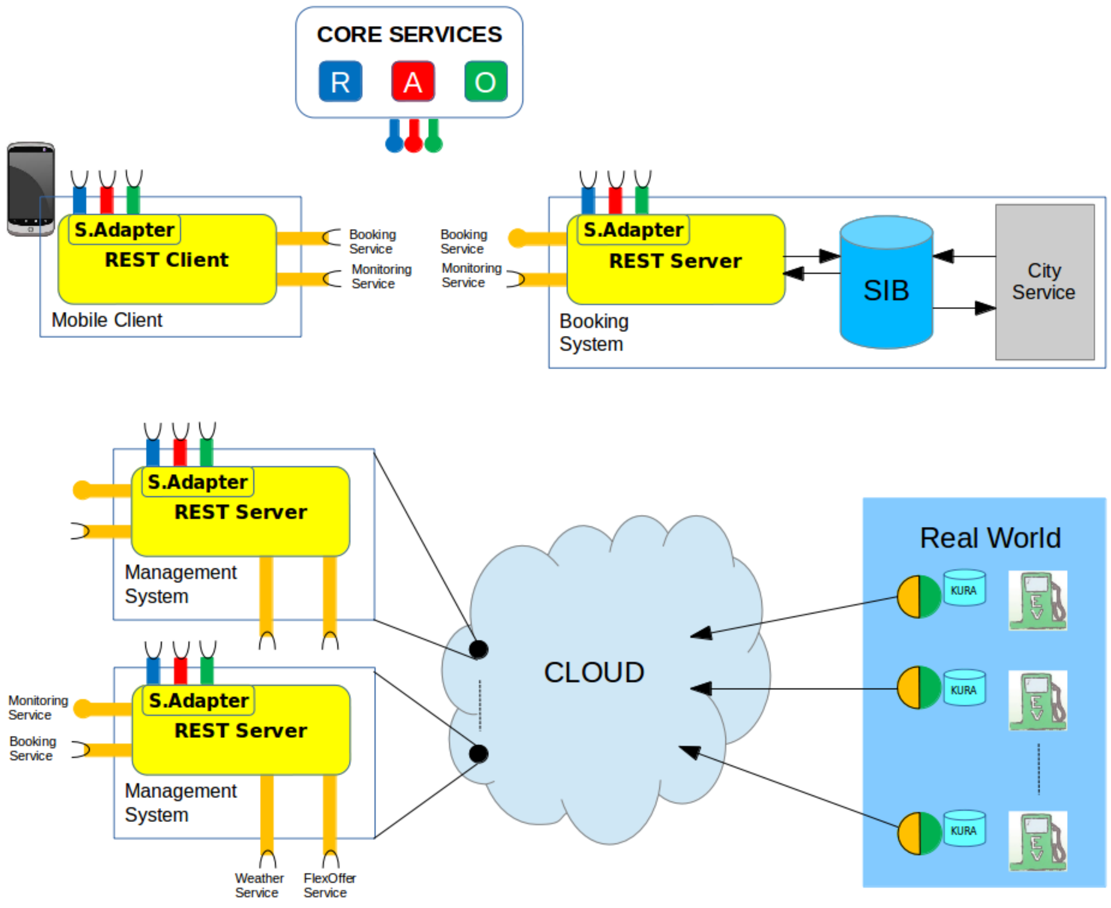


Figure 5.1: Arrowhead architecture scenarios for the real world.

Service. It instantiates a Management System for each EVSE, which consumes in turn the Booking System periodically to get informations about the existing reservations for the respective charging station. Furthermore, the system consumes the external services FlexOffer and weather forecast.

Booking Scenario

This scenario involves the Arrowhead Service Producer referred to as Booking System (BS), already defined. The scenario shows a custom arrowhead consumer, which may be represented by a real mobile application connected to a physical electrical vehicle, consuming the Booking Service. The scenario instantiates an unique BS which consumes in turn the Monitoring Service to be always updated about the condition and the status of the EVSEs.

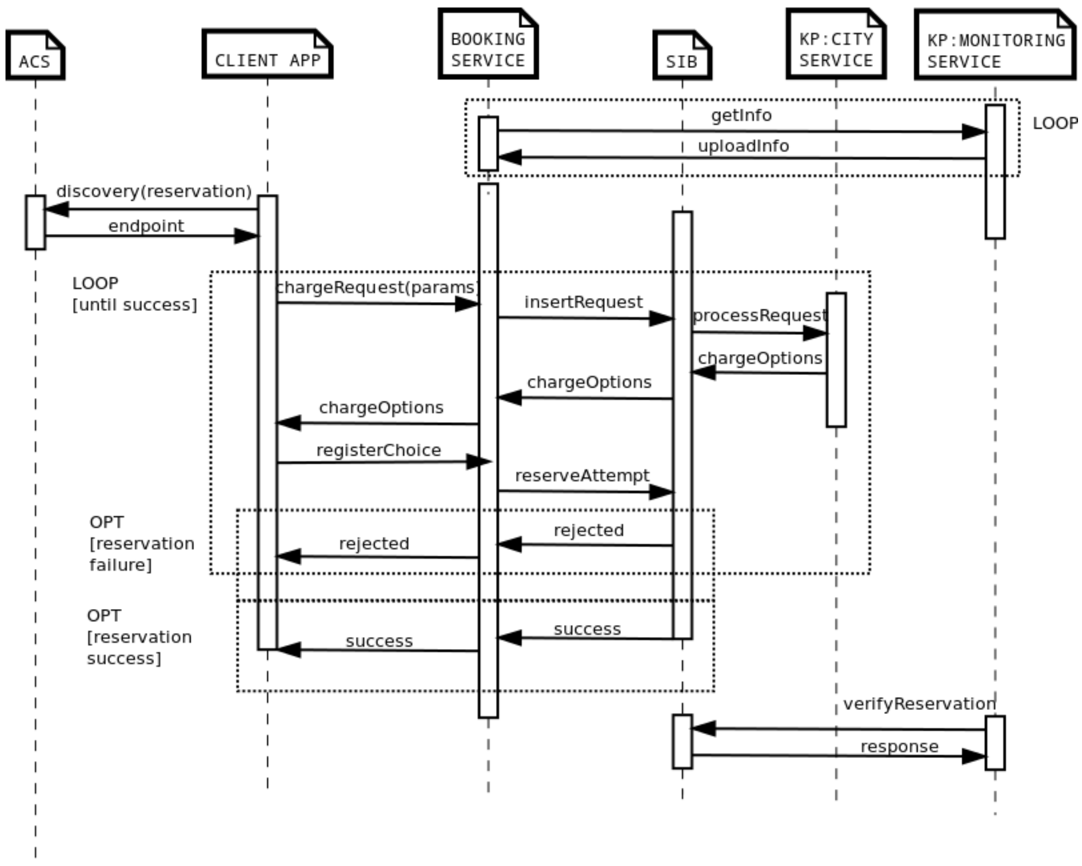


Figure 5.2: Complete reservation process.

Recharge Scenario

This scenario occurs when a vehicle arrives at a charging station and needs to recharge, hence consumes the Monitoring Service. The Management System responsible for that charging station, once called through the service, consumes in turn the Booking Service to know if that vehicle is reserved correctly for that moment. If so, the recharging process is possible, while the user may monitor the charge progress by consuming the Monitoring Service.

5.2 The Simulated Arrowhead E-Mobility Scenario

In this section is presented how WP9, within Task 9.3, designed a simulated scenario, the System-of-Systems developed for this thesis work, aiming to be parallel to the real scenario in section 5.1. As the real scenario design in WP3 is, as stated before, still ongoing, there are some discrepancies with the internal structure, although the Arrowhead-compliant part of the System-of-Systems should offer the same service interfaces to the extern.

5.2.1 Use Cases

The use cases designed for the simulated version of the scenario are, obviously, slightly different from the ones designed in the real world. At the moment, the system is not interfacing with any external pricing system (like FlexOffer) or weather forecasting service, however it needs to offer a vehicle monitoring service as well, because the simulated vehicles are not clearly monitorable directly form the end user through some kind of custom, short distance protocol (like Bluetooth). The use case extracted from this different scenario are the following:

1. The most important use case is given by the reservation facilities. An user must be able to interface with a booking system and perform a reservation submitting a preferred time and place and shall be able to choose among the offered possibilities which are given according to which charging station is not busy, reserved or faulted.

2. The end user must be able to monitor one of the simulated vehicles, get its data and perform booking requests impersonating it.

3. An user must be able to monitor the recharging process parameters while his or her vehicle is recharging at a public charging station.

4. The charging stations must provide information about their statuses and electrical parameters (those informations are used, within the simulation, by the tester to keep track of the charging station).

5. A charging station must be able to verify if an user is correctly reserved to start the recharging process.

5.2.2 Systems and Services

The above mentioned use cases leaded to the development of three main system concepts: the *Booking System*, the *EVSE Simulator Management System* and the *Vehicle Simulator Management System*.

The Booking System is developed as an exact parallel of the one presented in the real scenario, however it presents a completely different internal structure because, of course, it has to interact with simulated entities. In the scope of the legacy simulation system presented, the Booking System is an Arrowhead adapter combined with the City Service as a logical part. From an external point of view it provides the *Booking Service* (BS) as it does in the real world, exposing the same service interface to the extern, however it does not consume any Arrowhead service as the informations about the charging stations are obtainable directly from the City SIB. Hence it is a Service Producer.

The EVSE Simulator Management System is developed as an exact parallel of the Management System presented in the real world scenario. In the scope of the legacy simulation system presented, the EVSE Simulator Management System is an Arrowhead Adapter combined with the EVSEs running on the simulator. From an external point of view it provides the *EVSE Simulator Monitoring Service* (Esms), which is exactly the same as the Monitoring Service in the real world, however it does not consume any Arrowhead service as the informations about the reservations (to verify if a vehicle is reserved) are obtainable directly from the City SIB. Hence it is a Service Producer.

The Vehicle Simulator Management System is developed as a Management System for the vehicles, to provide all the parameters about the vehicle itself obtainable by its driver. In the scope of the legacy simulation system presented, the Vehicle Simulator Management System is an Arrowhead Adapter combined with the vehicles running on the simulator. From an external point of view, it provides the *Vehicle Simulator Monitoring Service* (Vsms), which is a Monitoring Service performing on vehicles. It does not consume any Arrowhead service, thus it is a Service Provider.

5.2.3 Retrievable Data

In this section is explained which data has been considered necessary to be retrievable by an Arrowhead client. Each data type and detail has been agreed with Centro Ricerche Fiat, the WP3 leader. The following table shows each data retrievable through the service provided, the actual service containing the information, the inputs to be passed, the data type received in output and the respective field in the ontology representing it. It has to be pointed out that the SOA protocol used, in the scope of the simulation, has been REST for all the services and that the Service Consumer aiming to consume the services described is the end user (or a simulation of it).

Data	**Provider**	**Input**	**Format**	**Ontology**
evseUri	Esms	-	string	EVSE

availability	Esms	-	string	EVSE-> hasAvailability
faultCode	Esms	-	string	EVSE-> hasFaultCode
chargingStatus	Esms	-	string	EVSE-> hasChargingStatus
maxEnergyCapability	Esms	-	double	EVSE-> hasMaxEnergyCapability
maxPowerCapability	Esms	-	double	EVSE-> hasMaxPower
priceData	Esms	-	double	ChargeProfile-> hasPrice
GPSposition	Esms	-	[double, double]	GCP-> hasGPSData
chargeProgress	Esms	-	struct: [starting time, energy recharged]	EVSE-> hasChargeProgress
chargeReport	Esms	-	struct: [starting time, energy recharged, ending time]	EVSE-> hasLastChargedEnergy
vehicleUri	Vsms	-	string	Vehicle
vehicleUsername	Vsms	-	string	Vehicle-> hasUser-> hasUserIdentifier
maxEnergyCapability	Vsms	-	double	BatteryData-> hasCapacity
maxPowerCapability	Vsms	-	double	BatteryData-> hasPower
stateOfCharge	Vsms	-	double	BatteryData-> hasStateOfCharge
GPSposition	Vsms	-	[double, double]	Vehicle-> hasGPSData
chargeProgress	Vsms	-	struct: [starting time, energy recharged]	EVSE-> hasChargeProgress
chargeReport	Vsms	-	struct: [starting time, energy recharged, ending time]	EVSE-> hasLastChargedEnergy
chargeRequest	BS	chargeRequest	chargeResponse	ChargeResponse-> hasRelatedRequest

reservationFind	BS	user-URI	reservation	Reservation
checkConfirm	BS	charge-Option	boolean	ChargeOption-> confirmBySystem
acknowledge	BS	charge-Option	boolean	ChargeOption-> ackByUser
reservationRetire	BS	res. ID	boolean	Reservation

5.2.4 Structure

As a big picture design, we can distinguish among three different main scenarios: the EVSE monitoring scenario, the vehicle monitoring scenario and the booking scenario. In this section all these scenarios are presented with a focus on how they are parallel to the ones presented in section 5.1.3. They can be considered separately as System-of-Systems, however, when they are in execution at the same time (as it would be in the real world), the System-of-Systems is given by the union of all the elements and the interactions involved.

The common entities to all of them are:

- The ACS, hosted, in our example, by the BnearIT VPN.
- The E-Mobility legacy simulator, which can be considered as a black box and simulates all the charging station and vehicles in the real world. The interactions from and to the extern are always mediated by the City SIB and the Dash SIB, so these elements are the only (legacy) connectors the system exposes. The City SIB acts both as the Cloud and the SIB hosted by the Booking System in the real scenario.
- An event injector, a common Knowledge Processor that can inject events (such as the charging station fault) in the SIB for testing purposes.

EVSE Monitoring Scenario

This scenario involves the Arrowhead Service Producer referred to as EVSE Simulator Management System (ESMS), already defined and explained in detail in section 5.3.1. The scenario shows a custom arrowhead consumer, which may be represented by a test desktop client application or the real mobile application, consuming the EVSE Simulator Monitoring Service. It instantiates a ESMS for each EVSE in the simulator, as shown in figure 5.3.

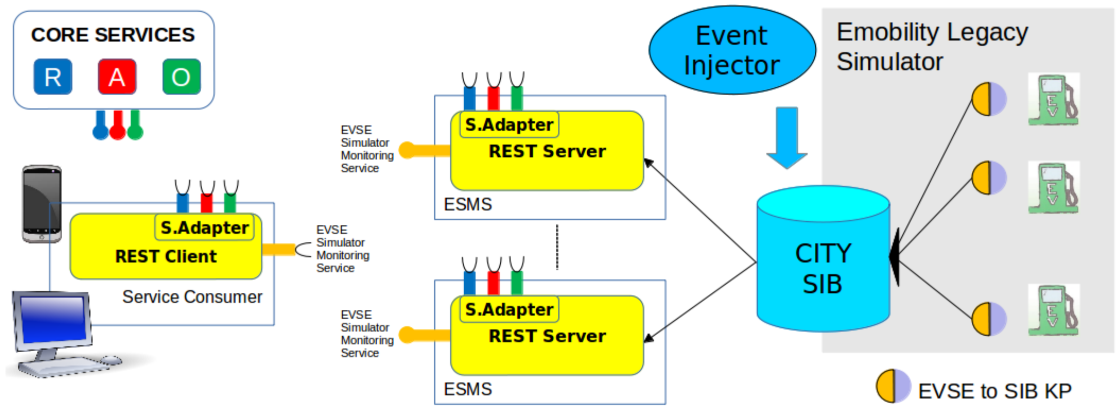


Figure 5.3: ESMS monitoring scenario.

Vehicle Monitoring Scenario

This scenario involves the Arrowhead Service Producer referred to as Vehicle Simulator Management System (VSMS), already defined and explained in detail in section 5.3.2. The scenario shows a custom arrowhead consumer, which may be represented by a test desktop client application or the real mobile application, consuming the Vehicle Simulator Monitoring Service. It instantiates a VSMS for each electrical vehicle in the simulator, as shown in figure 5.4.

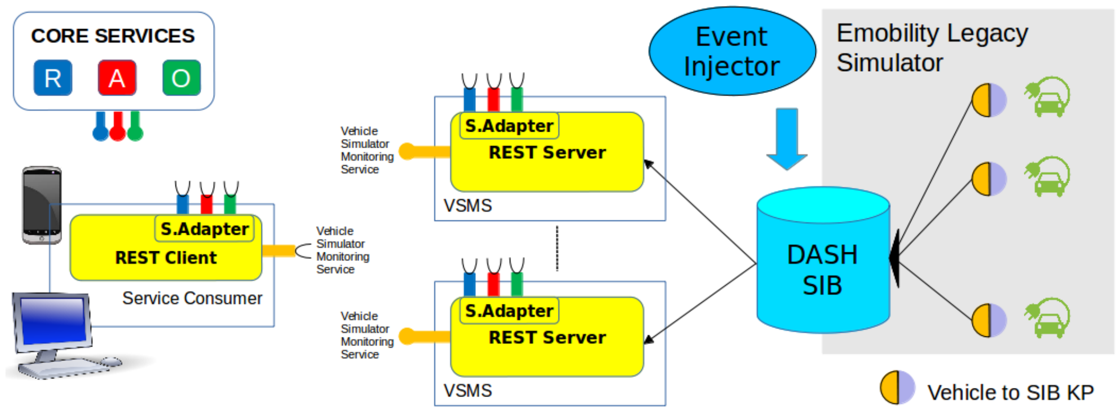


Figure 5.4: Vehicle monitoring scenario.

Booking Scenario

This scenario involves the Arrowhead Service Producer referred to as Booking System (BS), already defined and explained in detail in section 5.3.3. The scenario shows a custom arrowhead consumer, which may be represented by a test desktop client application or the real mobile application, consuming the Booking Service. To be able to consume the Booking Service, a consumer must monitor one of the vehicles, thus it is necessary (even if it is not shown in the figure) that the consumer itself is consuming the Vehicle Simulator Monitoring Service as well. The scenario instantiates an unique BS, as shown in figure 5.5.

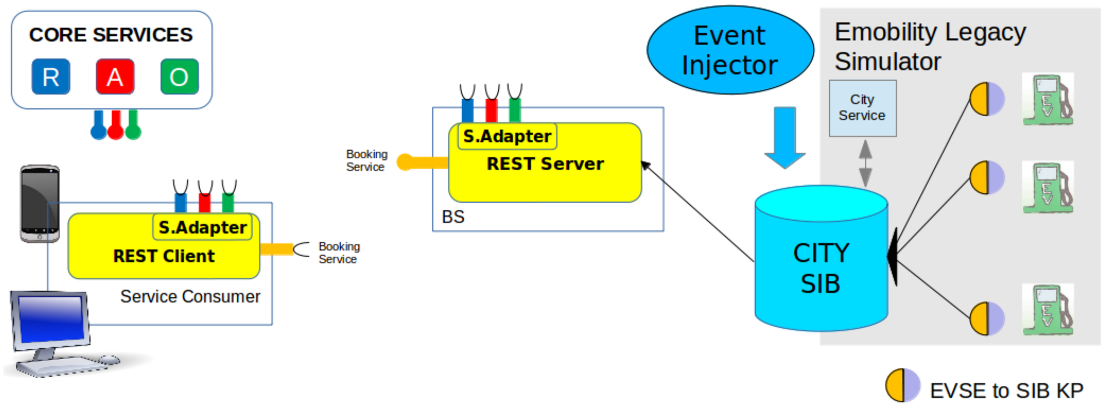


Figure 5.5: BS monitoring scenario.

5.3 Service Producers in the Simulation

This section details the Service Producers that were developed in the scope of this project as White Box, thus explaining the internal implementation. All of them were implemented as Compact Service producers using the APIs described in 4.

5.3.1 The EVSE Simulator Management System

This Service Producer is deployed in a unique program which instantiates a REST server for each charging station in the system. Once started, it searches the City SIB for each instance of an EVSE records and, after fetching the EVSE id, it publishes a different service for each of them, instantiating respectively a REST server to handle the requests from the extern. Each service is represented by the following record format: `evseID._evse-ws-http._tcp.srv.test.bnearit.arrowhead.eu:progressivePort/monitor`, where evseID stands for the actual EVSE Id fetched from the City SIB, while progressivePort stands for a port generated progressively starting from 20010 (in our particular example). The endpoint where the service is retrievable will then be `serverHostname:progressivePort/monitoring`. Data are physically retrievable adding a sub-path at the end of the endpoint:

`/evseUri`	HTTP GET
`/availability`	HTTP GET
`/faultCode`	HTTP GET
`/chargingStatus`	HTTP GET
`/maxEnergyCapability`	HTTP GET
`/maxPowerCapability`	HTTP GET
`/priceData`	HTTP GET
`/GPSposition`	HTTP GET
`/chargeProgress`	HTTP GET
`/chargeReport`	HTTP GET

5.3.2 The Vehicle Simulator Management System

This Service Producer is deployed in a unique program which instantiates a REST server for each vehicle in the system. Once started, it searches the Dash SIB for each instance of an Vehicle records and, after fetching the vehicle id, it publishes a different service for each of them, instantiating respectively a REST server to handle the requests from the extern. During the process' life cycle, the Dash SIB is being monitored constantly to get whether any new vehicle is created or any vehicle is deleted through a subscription. When any of those events happen, the program automatically creates a new service for the newly spawned vehicle or erases the service for the deleted vehicle respectively. Each service is represented by the following record format: `vehicleID._vehicle-ws-http._tcp.srv.test.bnearit.arrowhead.eu:progressivePort/monitor`, where vehicleID stands for the actual vehicle Id fetched from the Dash SIB, while progressivePort stands for a port generated progressively starting from 21010 (in our particular example). The endpoint where the service is retrievable will then be `serverHostname:progressivePort/monitoring`. Data are physically retrievable adding a sub-path at the end of the endpoint:

`/vehicleUri`	HTTP GET
`/vehicleUsername`	HTTP GET
`/maxEnergyCapability`	HTTP GET
`/maxPowerCapability`	HTTP GET
`/stateOfCharge`	HTTP GET

`/GPSposition`	HTTP GET
`/chargeProgress`	HTTP GET
`/chargeReport`	HTTP GET

5.3.3 The Booking System

This Service Producer is deployed in a unique program which instantiates a unique REST server for the whole system. Once started, it instantiates the connection to the City SIB, as well as a single service, denoted by the record `BookingSystem._reservation-ws-http._tcp.srv.test.bnearit.arrowhead.eu:40010/bridge`, where the port 40010 is hard coded in our particular example. The endpoint where the service is retrievable will then be `serverHostname:40010/bridge`. Data are physically retrievable adding a sub-path at the end of the endpoint:

`/chargeRequest`	HTTP POST	requires a chargeRequest as a Form in input.
`/reservationFind`	HTTP POST	requires a user URI in input.
`/checkConfirm`	HTTP POST	requires a chargeOption URI in input.
`/acknowledge`	HTTP POST	requires a chargeOption URI in input.
`/reservationRetire`	HTTP POST	requres a reservation URI in input.

Chapter 6

Arrowhead Service Test Consumer

6.1 Overview

The Test consumer, developed to test the different functions offered by the global E-Mobility simulated scenario, has been implemented in parallel with the development of the Service Providers presented in section 5.3. Hence, it uses the exact same version of the layer 1 API and implements a different Service Consumer class for each of the Service Providers that it is interfacing to. In particular, it implements a layer 2 API consisting of the following classes, all of them extending the class *ArrowheadServiceConsumer*:

- *EmobilityEvseConsumerREST_WS*: this class implements a consumer for the EVSE Simulator Management Service, thus it focuses on the function *consumeResource(String path)*, leaving the other two versions empty. The reason for this is that every single call to the above mentioned service is an HTTP GET without parameters.
- *EmobilityVehicleConsumerREST_WS*: this class is specular to the previous one. It implements a consumer for the Vehicle Simulator Management Service and, for the same reason as above, it implements only the function *consumeResource(String path)* for HTTP GET requests.
- *EmobilityReservationConsumerREST_WS*: this class implements a consumer for the Booking Service. Since nearly all the calls to such service are HTTP POST requests including parameters (as specified in subsection 5.3.3), this consumer focuses on the implementation of the function *consumeResource(String path, Form form)*, using Java built-in Form classes to encapsulate the parameters.

The data exchanged with the servers are encapsulated in XML envelopes, however the developer shall never interact with the XML facilities because the Java

REST Webservice API includes a mapping XML-to-class. In this way, the Service Provider needs to pass the whole object as a response, the Java REST Webservice library maps it to an XML envelope and maps it back to an object on the client's side. For this reasons it is clear that both the client and the server were more likely to be implemented in the same language, in order to use these facilities.

The GUI interface has been built using the Java Swing libraries together with Eclipse's *WindowBuilder* tool.

6.2 Functions

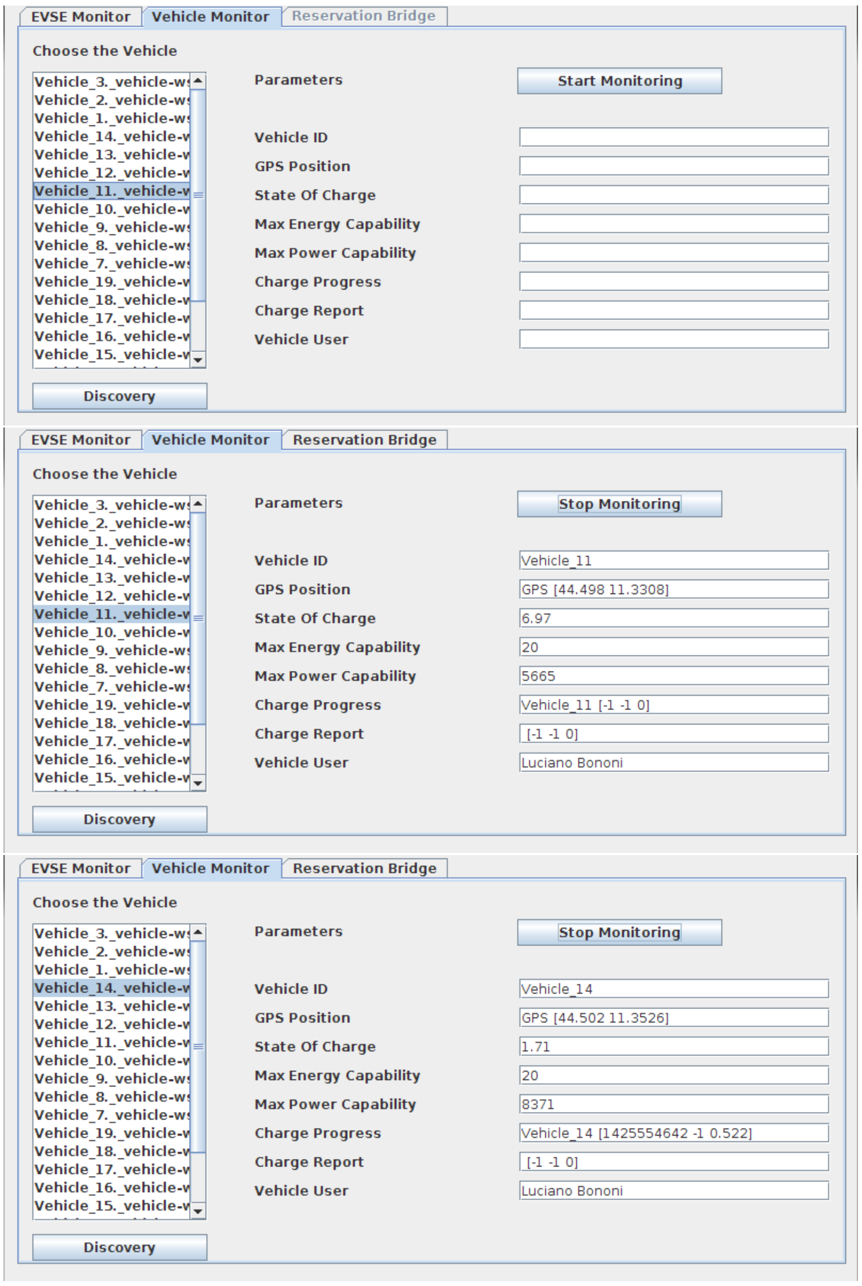
The GUI offers to the end user a tab-based view in which he or she can perform different Arrowhead-compliant actions against the three respective Service Providers. In this section we show the main functions callable from the three tab panels.

6.2.1 Monitoring a Vehicle

The central tab, called "Vehicle Monitor", handles all the communications with the Vehicle Simulator Management System. The main functionalities are shown in figure 6.1. Pushing the "Discovery" button will display a list of the vehicle services published on the Service Registry. This operation is performed by calling a discovery function which filters services by type. After selecting one of the vehicles it is possible to monitor it by pushing the "Start Monitoring" button, which will change in "Stop Monitoring" once pressed. This will trigger the instantiation of a new thread aiming to cycling over a set of consuming calls. In particular, the thread will call every single consuming function against the VSMS in order to fill the fields shown in figure 6.1 one by one and it stops when the "Stop Monitoring" button is pushed.

6.2.2 Monitoring a Charging Station

The left tab, called "EVSE Monitor", handles almost all the communications with the EVSE Simulator Management System. The main functionalities are shown in figure 6.2 and it is noticeable how this tab shows parallelism with the previous one. Indeed it provides nearly the same functionalities: pushing the "Discovery" button gets all the EVSE services from the Service Registry, pushing the "Start Monitoring" button starts a thread which performs a sequence of consuming calls and fills the respective fields and the "Stop Monitoring" button stops the thread. It is noticeable how the GUI grouped the availbility, faultCode and chargingStatus (in this order) in the same field, even though the calls are performed separately.



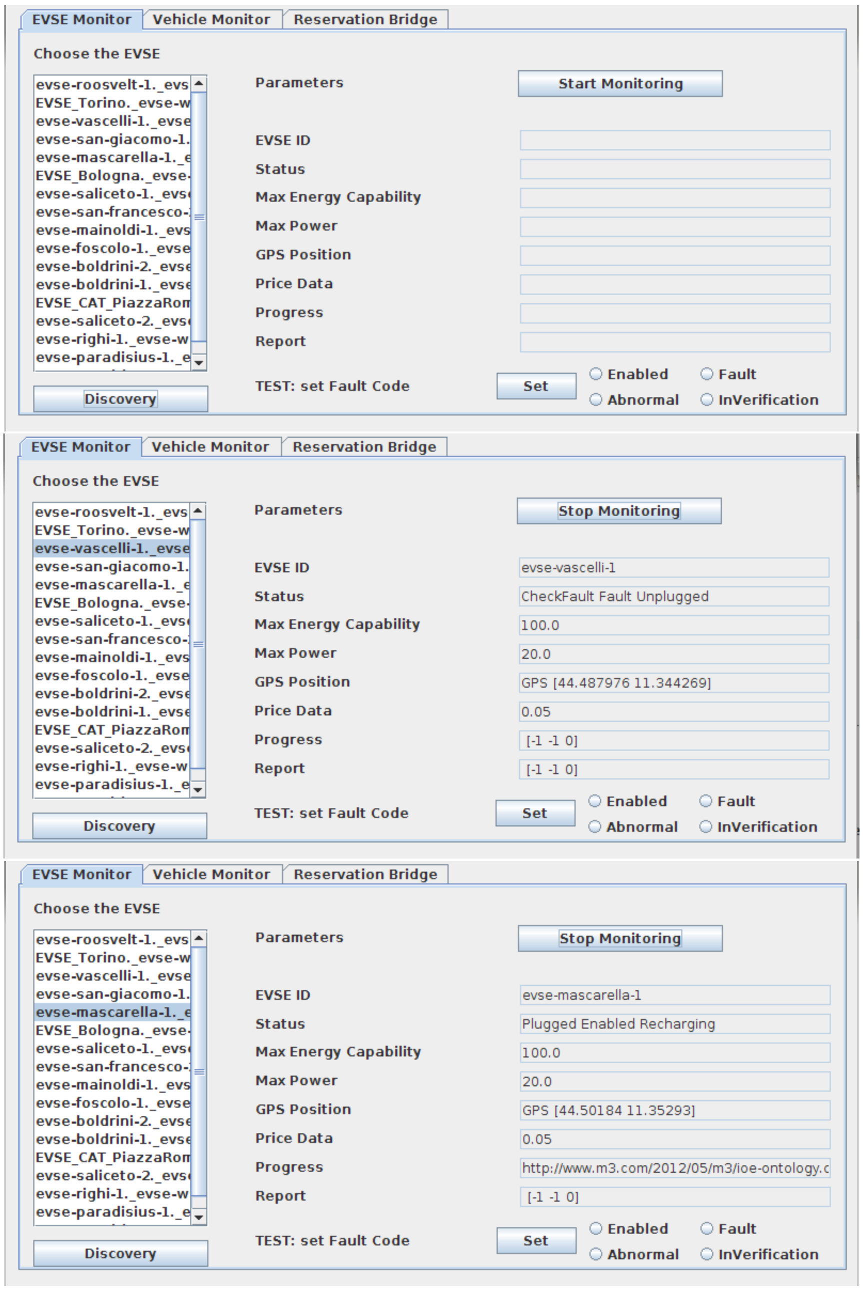
Vehicle_3._vehicle-w
Vehicle_2._vehicle-w
Vehicle_1._vehicle-w
Vehicle_14._vehicle-v
Vehicle_13._vehicle-v
Vehicle_12._vehicle-v
Vehicle_11._vehicle-v
Vehicle_10._vehicle-v
Vehicle_9._vehicle-w
Vehicle_8._vehicle-w
Vehicle_7._vehicle-w
Vehicle_19._vehicle-v
Vehicle_18._vehicle-v
Vehicle_17._vehicle-v
Vehicle_16._vehicle-v
Vehicle_15._vehicle-v

Parameters	(top)	(center)	(bottom)
Vehicle ID		Vehicle_11	Vehicle_14
GPS Position		GPS [44.498 11.3308]	GPS [44.502 11.3526]
State Of Charge		6.97	1.71
Max Energy Capability		20	20
Max Power Capability		5665	8371
Charge Progress		Vehicle_11 [-1 -1 0]	Vehicle_14 [1425554642 -1 0.522]
Charge Report		[-1 -1 0]	[-1 -1 0]
Vehicle User		Luciano Bononi	Luciano Bononi

Figure 6.1: Vehicle Monitor tab. After performing a discovery (top), after monitoring a vehicle (center), after monitoring a vehicle under recharge (bottom)

6.2.3 Performing Reservations

The right tab, called “Reservation Bridge”, is responsible for the reservation process and it handles all the communications with the Booking System. The charge


EVSE Monitor | Vehicle Monitor | Reservation Bridge

Choose the EVSE

evse-roosvelt-1._evs
EVSE_Torino._evse-w
evse-vascelli-1._evse
evse-san-giacomo-1.
evse-mascarella-1._e
EVSE_Bologna._evse-
evse-saliceto-1._evs
evse-san-francesco-
evse-mainoldi-1._evs
evse-foscolo-1._evse
evse-boldrini-2._evse
evse-boldrini-1._evse
EVSE_CAT_PiazzaRon
evse-saliceto-2._evs
evse-righi-1._evse-w
evse-paradisius-1._e

Discovery

Parameters — Start Monitoring

EVSE ID	
Status	
Max Energy Capability	
Max Power	
GPS Position	
Price Data	
Progress	
Report	

TEST: set Fault Code — Set — Enabled / Fault / Abnormal / InVerification

EVSE Monitor | Vehicle Monitor | Reservation Bridge

Choose the EVSE

evse-roosvelt-1._evs
EVSE_Torino._evse-w
evse-vascelli-1._evse
evse-san-giacomo-1.
evse-mascarella-1._e
EVSE_Bologna._evse-
evse-saliceto-1._evs
evse-san-francesco-
evse-mainoldi-1._evs
evse-foscolo-1._evse
evse-boldrini-2._evse
evse-boldrini-1._evse
EVSE_CAT_PiazzaRon
evse-saliceto-2._evs
evse-righi-1._evse-w
evse-paradisius-1._e

Discovery

Parameters — Stop Monitoring

EVSE ID	evse-vascelli-1
Status	CheckFault Fault Unplugged
Max Energy Capability	100.0
Max Power	20.0
GPS Position	GPS [44.487976 11.344269]
Price Data	0.05
Progress	[-1 -1 0]
Report	[-1 -1 0]

TEST: set Fault Code — Set — Enabled / Fault / Abnormal / InVerification

EVSE Monitor | Vehicle Monitor | Reservation Bridge

Choose the EVSE

evse-roosvelt-1._evs
EVSE_Torino._evse-w
evse-vascelli-1._evse
evse-san-giacomo-1.
evse-mascarella-1._e
EVSE_Bologna._evse-
evse-saliceto-1._evs
evse-san-francesco-
evse-mainoldi-1._evs
evse-foscolo-1._evse
evse-boldrini-2._evse
evse-boldrini-1._evse
EVSE_CAT_PiazzaRon
evse-saliceto-2._evs
evse-righi-1._evse-w
evse-paradisius-1._e

Discovery

Parameters — Stop Monitoring

EVSE ID	evse-mascarella-1
Status	Plugged Enabled Recharging
Max Energy Capability	100.0
Max Power	20.0
GPS Position	GPS [44.50184 11.35293]
Price Data	0.05
Progress	http://www.m3.com/2012/05/m3/ioe-ontology.c
Report	[-1 -1 0]

TEST: set Fault Code — Set — Enabled / Fault / Abnormal / InVerification

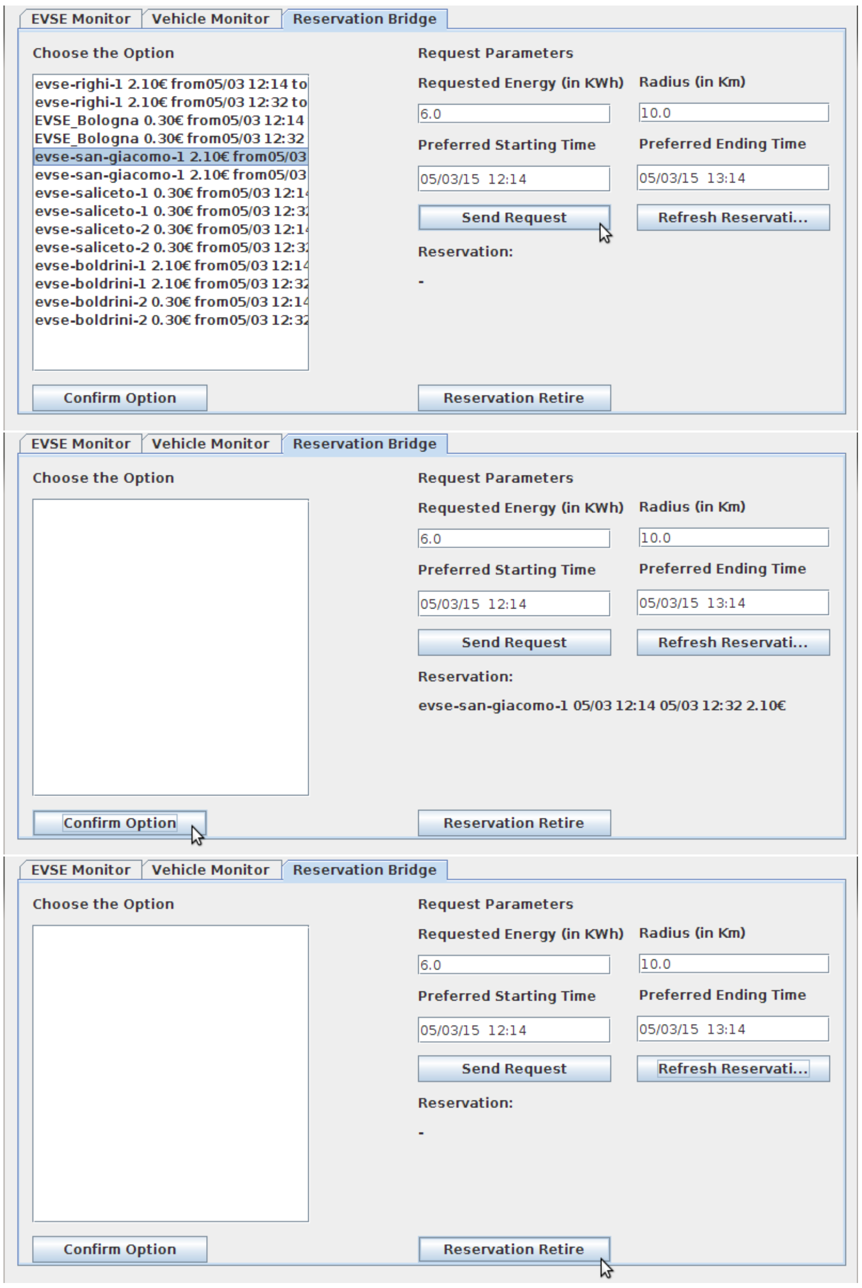
Figure 6.2: EVSE Monitor tab. After performing a discovery (top), after monitoring a faulted EVSE (center), after monitoring an EVSE while recharging (bottom)

request is performed after the end user selected the parameters to fit his or her needs, as can be seen in figure 6.3. Once the "Send Request" button is pushed, the systems sends a charge request to the Booking System and gets a set of charge options back, displayed in the list on the left. Charge options have been chosen to be displayed including EVSE ID, starting time, ending time and price, but this is highly customizable. The next step is choosing a charge option and clicking on “Confirm Option”. This action performs two steps, firstly it sets the confirm by user and checks the confirm by system and then it sends an acknowledgement by user. Normally the acknowledgement is always confirmed, unless connection problems come up. If the confirm by system is negative, then the user is forced to start over the process. Once the acknowledgement has been sent, the actual reservation is displayed as shown. At this point the user can either push “Refresh Reservation”, with which it gets the reservation (useful after an application reboot), or perform a reservation retire with the proper button.

The Fault Detector

Getting a reservation confirmed, in our test application, triggers a side-effect. Once got the reservation confirmation, the application immediately starts a thread in background which checks the status of the reserved EVSE. If this status changes to a kind of fault, the application pops up an alert, as can be seen in figure 6.4, as well as when the charging station changes its status from faulted to enabled. The test application provides a non-Arrowhead-compliant event injector which can mutate an EVSE fault code from the EVSE Monitor tab. This is an exception to the normal interaction, in fact, in this case, the test application directly connects to the City SIB.

In the scenario architectures presented in section 5.2.4, this tool may be seen as the “Event Injector”.



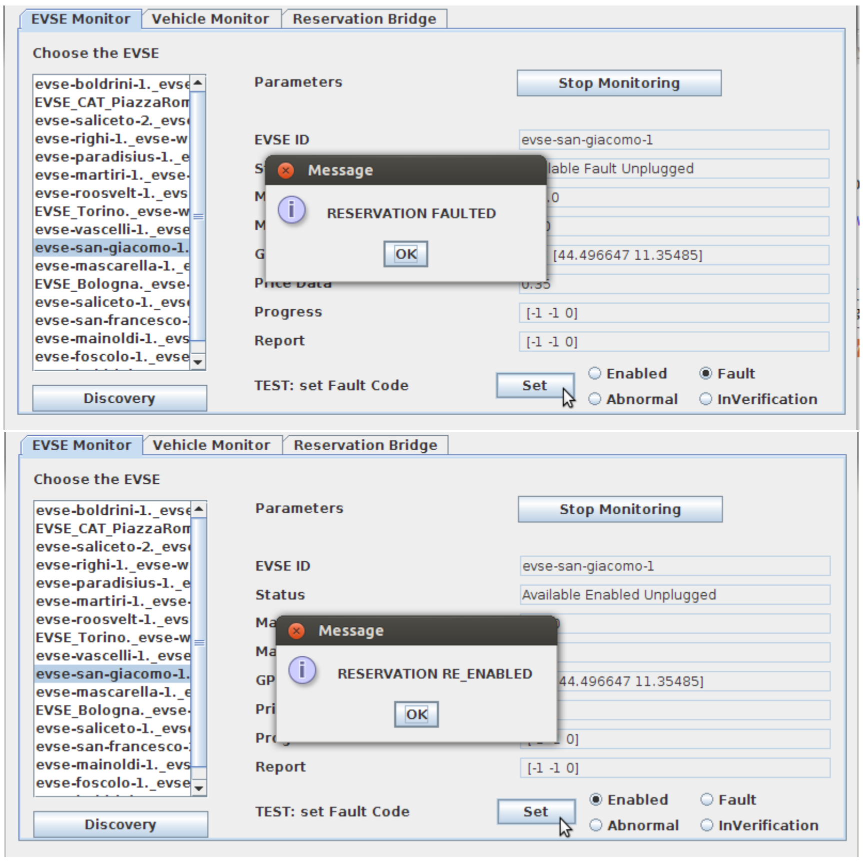
evse-righi-1 2.10€ from05/03 12:14 to
evse-righi-1 2.10€ from05/03 12:32 to
EVSE_Bologna 0.30€ from05/03 12:14
EVSE_Bologna 0.30€ from05/03 12:32
evse-san-giacomo-1 2.10€ from05/03
evse-san-giacomo-1 2.10€ from05/03
evse-saliceto-1 0.30€ from05/03 12:1
evse-saliceto-1 0.30€ from05/03 12:3
evse-saliceto-2 0.30€ from05/03 12:1
evse-saliceto-2 0.30€ from05/03 12:3
evse-boldrini-1 2.10€ from05/03 12:14
evse-boldrini-1 2.10€ from05/03 12:32
evse-boldrini-2 0.30€ from05/03 12:14
evse-boldrini-2 0.30€ from05/03 12:32

Requested Energy (in KWh): 6.0
Radius (in Km): 10.0
Preferred Starting Time: 05/03/15 12:14
Preferred Ending Time: 05/03/15 13:14
Reservation: -

Requested Energy (in KWh): 6.0
Radius (in Km): 10.0
Preferred Starting Time: 05/03/15 12:14
Preferred Ending Time: 05/03/15 13:14
Reservation:
evse-san-giacomo-1 05/03 12:14 05/03 12:32 2.10€

Requested Energy (in KWh): 6.0
Radius (in Km): 10.0
Preferred Starting Time: 05/03/15 12:14
Preferred Ending Time: 05/03/15 13:14
Reservation: -

Figure 6.3: Booking System tab. After performing a request (top), after selecting an option (center), after retiring a reservation (bottom)



evse-boldrini-1._evse
EVSE_CAT_PiazzaRom
evse-saliceto-2._evse
evse-righi-1._evse-w
evse-paradisius-1._e
evse-martiri-1._evse-
evse-roosvelt-1._evs
EVSE_Torino._evse-w
evse-vascelli-1._evse
evse-san-giacomo-1.
evse-mascarella-1._e
EVSE_Bologna._evse-
evse-saliceto-1._evse
evse-san-francesco-
evse-mainoldi-1._evs
evse-foscolo-1._evse

EVSE ID: evse-san-giacomo-1
S lable Fault Unplugged
M 0
G [44.496647 11.35485]
Price Data 0.35
Progress [-1 -1 0]
Report [-1 -1 0]

RESERVATION FAULTED

TEST: set Fault Code — Enabled [✓ Fault] Abnormal InVerification

EVSE ID: evse-san-giacomo-1
Status: Available Enabled Unplugged
GP 44.496647 11.35485]
Pr 0]
Report [-1 -1 0]

RESERVATION RE_ENABLED

TEST: set Fault Code — [✓ Enabled] Fault Abnormal InVerification

Figure 6.4: After injecting fault in a reserved EVSE (top), after injecting availability in a reserved EVSE (center).

Chapter 7

Conclusions

At the end of this experience, which took place over a span of more than two years, the whole UniBO team and I brought a consistent contribution to the Arrowhead project in terms of demonstration platforms. We demonstrated how a charging infrastructure within a single city can tolerate a certain amount of electrical vehicles to satisfy the users' demand for recharge. We demonstrated how the Arrowhead Framework can constitute an efficient bottleneck for the communication between heterogeneous systems within the scope of different scenarios and use cases. We demonstrated how a client application should be developed in order to provide the end user with an efficient set of functionalities. Hence, we can say that most of the goals we specified have been reached in a sufficient measure. Clearly there is still a lot of work to do, both in the scope of the simulator itself and in the scope of the Arrowhead Project, indeed such project is still ongoing and it recently entered in its second generation (the project partners expect at least a third generation).

Future Work

As specified, a lot of different features have been implemented, but we found a lot of improvements (which are demonstrated to be possible with significantly low effort) that we are planning to carry out in the future:

- Port the client demo application on a mobile device to simulate the true user experience.
- Refine the ontology deleting all the obsolete references.
- Implement support for different SOA protocols in the Arrowhead API.
- Refine the interaction with Orchestration and Authorisation core services.
- Deploy customized ACS within a local network.

- Re-implement the “vehicle control”, for which the vehicles in the simulation are not following their normal activity flow while controlled from an external device. After the Remote Monitoring Mobile Application has been considered obsolete, the function has been disposed.
- Standardize the service name within the Arrowhead Service Matrix.
- Implement both the publishing and the usage of metadata in the service records.

Appendices

.1 Local Core Services

The developers of the ACS provided all the Arrowhead Partners with an operating system image (an ad-hoc CentOS .iso file) which is able to host by default the ACS and a DNS server. The download instruction and the installation manual are available at `https://forge.soa4d.org/svn/arrowhead/CommonDesignRepository/03.APPROVED/05.Prototypes/CoreSystems1.0/`. The installation manual lacks some description about common errors, one of which regards the secure consumption of the services, which requires basic knowledges about certificates. This problem has not been solved at the moment, it is possible, however, to consume non-secure services, provided that the DNS created within the CentOS system is reachable through the `/etc/hosts` file (not specified in the guide) and such a system must be able to reach all the hosts providing and consuming services through DNS lookup and vice versa.

Alternatively, it is possible to use the ACS installed in the BnearIT servers, which are fully up and running without certificate issues. In order to connect to them, it is necessary to connect to BnearIT's VPN through the software SoftEther [56] and follow the instructions available at `https://forge.soa4d.org/svn/arrowhead/CommonDesignRepository/03.APPROVED/06.Governance/Testbed/ATFG1TestLab/`.

To test if the ACS are effectively reachable it is possible to download a Hello World application, written using the API introduced in this project, at `https://forge.soa4d.org/svn/arrowhead/WP9/Task9.3/Working/UniBO/`.

.2 Environment Installation

After being sure that the ACS are up and running, we can proceed with the installation of the whole environment (this installation has been tested on Debian and Ubuntu operating systems, in particular, Debian 7 and Ubuntu 12.04 and 14.04). This environment installation brief manual has only slightly being updated from the one in [24].

Preliminary Libraries

Run the following command to install the necessary libraries:

```
sudo apt-get install bison flex build-essential zlib1g-dev tk8.4-dev
    blt-dev libxml2-dev libpcap0.8-dev autoconf automake libtool
    libxerces-c2-dev libproj-dev libproj0 libfox-1.6-dev libgdal1h
    libboost-dev
```

IDEs, Tools and Systems

It is necessary to install the OMNeT++ IDE (the version used for this project is 4.5) from `http://www.omnetpp.org/omnetpp/cat_view/17-downloads/1-omnet-releases`. Follow the instructions on the manual for the installation.

It is necessary to download, compile and install the Sumo simulator directly from the dedicated Sourceforge page (the version used in this demonstration is 0.21.0, older and newer versions may not work properly). It is retrievable at `http://sourceforge.net/projects/sumo/files/sumo/version0.21.0/` and it is provided with an installation manual.

It is necessary to download, compile and install the UniBO modified version of the Smart-M3 environment, in this case RedSIB version 0.9 (probably subsequent version may work as well). It is retrievable at `http://sourceforge.net/projects/smart-m3/files/Smart-M3-RedSIB_0.9/` and it is provided with an installation manual.

The simulator uses a third-party C library to implement the SSAP operations against the SIB: the *KPI Low*. They use in turn a third-party library for the XML parsing called Scew and retrievable at `http://nongnu.askapache.com/scew/scew-1.1.3.tar.gz`. Those libraries have been modified over time by me and Simone Rondelli to add SPARQL support, so they are retrievable in the project root.

The Main System-of-Systems

The legacy simulator is obtainable from the BitBucket repository `https://bitbucket.org/InternetOfEnergy/internet-of-energy` (branch "simulator-arrowhead") under clearance, as it is not open. From the project root (defined `ROOT/`) the KPI Low modified libraries are retrievable at `ROOT/kpi_low_mod` together with in installation manual.

From here, we can then import the simulation in the OMNeT++ IDE, setting as the workspace root folder the directory `ROOT/simulator` and importing the project "veins-2.1" from that same directory. It will be possible then building the simulation with a simple click, while the source code will be available at `ROOT/simulator/veins-2.1/examples/veins/`. In that folder there is a ready script, called totalScript, which starts all the components at a time: two SIBs, all the SIB's TCP listeners, the Java City Service and the simulation (you can ann the option –gui inside the script in the line starting with "./start-ioe" to have the Sumo GUI). It requires four parameters: the OMNeT++ configuration name, the current iteration, the number of total iterations and the Sumo Tools root folder; an example could be:

```
./totalScript.sh arrowhead 0 1 ~/Programs/sumo-0.21.0/tools/
```

IMPORTANT: it is required lo launch the script `ROOT/sync/synchronizer.py` in parallel as other simulations may occur at the same time.

After the simulation loaded all the EVSEs, it is possible to launch the three Service Providers:

```
java -jar ROOT/arrowhead/EVSESimulatorMonitoringSystem.jar
java -jar ROOT/arrowhead/VehicleSimulatorMonitoringSystem.jar
java -jar ROOT/arrowhead/BookingSystem.jar
```

If any service has been left published after closing all the providers, it is possible to launch our “vacuum cleaner” to erase them:

```
java -jar ROOT/arrowhead/EMVacuum.jar
```

.3 Acknowledgements

My part in the development of this project was long, with some interruptions I worked on it for its whole duration and it would never have been possible without a lot of help from many people. I spent the last two months of work as a guest in Luleå Tekniska Universitet and it has been one of the best experiences in my whole life.

I have to thank especially who, in Luleå, followed my project and helped me. First of all my supervisor, prof. Jerker Delsing, who, even if his agenda was incredibly packed, he always found time to follow my progress. Together with him my special thanks go to the other people in the department working in the Arrowhead project: Jens, Arash, Denis, Hasan and Pablo. Furthermore, thanks to the people from BnearIT, who came multiple times at my office to help me: Fredrik, Per and Thorsten.

Finally, here in Sweden, I had the possibility to meet awesome people, who, in a way, changed a little part of my life. Carsten, Linus, Paul, Charles, Daniel, Peter, Mitch, David, Norbert, Maria Isabelle, Laura, Marta, Stefan, Kerr, Riley, Chris, Bettina, Florian, Ariane, Michela, Nicoletta, Maria, Lara, Miguel, Hann, Sergio, Marcus, Andreas, Niklas and Maria: thanks to you all for having been a part of my life.

Non posso non ringraziare i miei professori, Tullio e Luciano. Hanno creduto in me e si sono affidati a me per questo progetto e senza di loro non avrei mai potuto fare questa esperienza che ha cambiato la mia vita. Assieme a loro non posso mancare di ringraziare Luca, Marco, Fabio, Riccardo e soprattutto Alfredo, che è stato il mio tramite per tutto questo periodo e ha lavorato con me e sul mio materiale una grande quantità di tempo.

Credo che il ringraziamento più profondo vada ai miei genitori, a mio fratello e a Mandi. È solo grazie a queste persone che ho potuto affrontare questa esperienza, al sostegno che mi hanno dato in tutti questi anni e che mi continuano a dare anche se sono lontano da casa. In un certo senso, sono la parte migliore di me.

Thank you. Grazie. Tack. Teşekkürler.

Bibliography

[1] Ciancarini, P., *Architectural Styles for Clouds and Services*, University of Bologna, lectures.

[2] Rawson, M., Kateley, S. *Electric Vehicle Charging Equipment Design and Health and Safety Codes*, California Energy Commission, August 1998

[3] M. Weiser, *The Origins of Ubiquitous Computing Research at PARC in the late 1980s* , IBM, 1999

[4] Dey, A. K., Abowd, A. K., Towards, G. D., *A Better Understanding of Context and Context-Awareness*, CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness, 2000

[5] Varga, A., *The OMNeT++ Discrete Event Simulation System*, `http://www.omnetpp.org`, European Simulation Multiconference (ESM'2001), Prague, Czech Republic. 2001.

[6] Cook, D., Das, S., *Smart Environments: Technology, Protocols and Applications* , Wiley Interscience, November 2004.

[7] Honkola, J., Laine, H., Brown, R., Trykkö, O. *Smart-M3 Interoperability Platform*, Nokia Research Center, Helsinki, Finland

[8] Wegener, A., et al. *TraCI: an Interface for Coupling Road Traffic and Network Simulators*, Proceedings of the 11th Communications and Networking Simulation Symposium. ACM. 2008, pp. 155–163.

[9] Köpke, A. et al. *Simulating Wireless and Mobile Networks in OMNeT++: the MiXiM vision*, Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops. ICST (Institute for Computer Sciences, Social-Informatics e Telecommunications Engineering), 2008, p. 71.

[10] Velte, A. T., *Cloud Computing: A Practical Approach*, 2010 McGraw Hill. ISBN 978-0-07-162694-1.

[11] Ovaska, E., Toninelli, A., Salmon Cinotti, T., *The Design Principles and Practices of Interoperable Smart Spaces*, Advanced Design Approaches to Emerging Software Systems, 2011

[12] Behrisch, M. et al. *Sumo-simulation of urban mobility-an Overview*, SIMUL 2011, The Third International Conference on Advances in System Simulation. 2011, pp. 55–60.

[13] Sommer, C., German, R., Dressler, F., *Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis*, IEEE Transactions on Mobile Computing 10.1, pp. 3–15. doi: 10.1109/TMC.2010.133. January 2011.

[14] Bedogni, L., Bononi, L., Di Felice, M., *Dynamic Backbone for Fast Information Delivery in Vehicular ad-hoc Networks: an Evaluation Study*, Proceedings of the 8th ACM Symposium on Performance Evaluation of Wireless ad hoc, Sensor, and Ubiquitous Networks. ACM. 2011, pp. 1–8.

[15] Oliveros, E. et al., *Web service Specifications Relevant for Service Oriented Infrastructures, Achieving Real-Time in Distributed Computing: From Grids to Clouds*, 2012 IGI Global, pp. 174–198, doi:10.4018/978-1-60960-827-9.ch010

[16] Ivanov, I., Van Sinderen, M., Shishkov, B., *Cloud Computing and Services Science*, Springer Science, 2012.

[17] Montori, F., *Project and Evaluation of an Experimental Platform about Internet of Energy for Electrical Vehicles*, Computer Science Bachelor Dissertation, `http://amslaurea.unibo.it/3900/`, July 2012.

[18] Bedogni L. et al., *Machine-to-Machine Communication over TV White Spaces for Smart Metering Applications*, Computer Communications and Networks (ICCCN), 2013 22nd International Conference on. IEEE. 2013, pp. 1–7.

[19] Bedogni, L., Bononi, L., Di Felice, M., D'Elia, A., Mock, R., Montori, F., Morandi, F., Roffia, L., Rondelli, S., Salmon Cinotti, T., Vergari, F., *An Interoperable Architecture for Mobile Smart Services over the Internet of Energy*, IEEE 14th International Symposium and

Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2013), Piscataway NJ, IEEE Press, 2013, pp. 1 - 6 (acts of: IEEE WoWMoM Workshop on Smart City and Ubiquitous Computing Applications (IEEE SCUCA 2013), Madrid, Spain, 2013.

[20] D'Elia, A. et al. *A Semantic Event Processing Engine Supporting Information Level Interoperability in Ambient Intelligence*, online `http://amsacta.unibo.it/3877/`, 2013

[21] Klisics, M., *Arrowhead SysD Service Registry DNS-SD*, version 1.0, Arrowhead Documentation, 2013.

[22] Klisics, M., *Arrowhead SysD Authorisation DNS-SD*, version 1.0, Arrowhead Documentation, 2013.

[23] Klisics, M., *Arrowhead SysD Orchestration DNS-SD*, version 1.0, Arrowhead Documentation, 2013.

[24] Rondelli, S., *A Framework of Analysis and Innovative Services for the Electrical Vehicles Mobility*, Computer Science Bachelor Dissertation, `http://amslaurea.unibo.it/6750/`, March 2014.

[25] *Arrowhead TA*, Contract shared among Arrowhead Partners, version 0.40, 2014.

[26] Blomstedt, F., *Arrowhead Cookbook*, version 1.1, Arrowhead Documentation, 2014.

[27] Ferreira, L. L., Zubia, M. C., Johansson, M., *Arrowhead Framework Definitions*, Arrowhead Documentation, 2014.

[28] Johansson, M., Mousavi, A., Kleyko, D., *Arrowhead interoperability Matrix*, Arrowhead Documentation, 2015.

[29] OASIS Group, *SOA Reference Model Definition*, `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm`

[30] Channabasavaiah, K., Holley, K., Tuggle, E. *Migrating to a service-oriented architecture*, IBM Developer Works Library, `http://www.ibm.com/developerworks/library/ws-migratesoa/`

[31] The open Group, *SOA Reference Architecture Technical Standard : Basic Concepts*, `http://www.opengroup.org/soa/source-book/soa_refarch/concepts.htm`

[32] Internet of Energy, http://www.artemis-ioe.eu/

[33] Arrowhead Partners, *The Arrowhead Project*, http://www.arrowhead.eu/about/

[34] *Advanced Research & Technology for EMbedded Intelligence and Systems*, http://www.artemis.eu/

[35] *Common Shared Arrowhead Repository*, https://forge.soa4d.org/svn/arrowhead/

[36] *Smart-M3 Official Website*, http://smart-m3.sourceforge.net/

[37] *SOFIA/ARTEMIS Project Page*, http://www.sofia-project.eu/

[38] *TotalFlex Project*, http://www.totalflex.dk/InEnglish/

[39] *KPI Low Libraries*, http://sourceforge.net/projects/kpilow/

[40] *Blue&Me*, http://www.blueandme.net/blueandme/index.aspx/

[41] Extensible Markup language, http://www.w3.org/XML/, referenced January 12^th 2015.

[42] Resource Description Framework, http://www.w3.org/RDF/

[43] Ontology Web Language, http://www.w3.org/OWL/

[44] Web services Description Language 1.1, http://www.w3.org/TR/wsdl, referenced January 12^th 2015.

[45] Simple Object Access Protocol 1.2, http://www.w3.org/TR/soap/, referenced January 12^th 2015.

[46] Constrained Application Protocol RFC7252, http://coap.technology/

[47] OPC Unified Architecture, https://opcfoundation.org/about/opc-technologies/opc-ua/

[48] Message Queue Telemetry Transport, http://mqtt.org/

[49] Extensive Message and Presence Protocol, http://xmpp.org/

[50] Data Distribution Services 1.2, http://www.omg.org/spec/DDS/1.2/

[51] Common Object Request Broker Architecture, `http://www.corba.org/`

[52] DNS Service Discovery, `http://www.dns-sd.org/`

[53] X.509 Certificates Standard, `https://www.ietf.org/rfc/rfc2459.txt`

[54] Jetty, `http://eclipse.org/jetty/`

[55] Jersey, `https://jersey.java.net/`

[56] SoftEther, `http://www.softether.org/`