Integrated CHOReOS middleware - Enabling large-scale, QoS-aware adaptive choreographies

Amira Ben Hamida, Fabio Kon, Nelson Lago, Apostolos Zarras, Dionysis Athanasopoulos, Dimitris Pilios, Panos Vassiliadis, Nikolaos Georgantas, Valérie Issarny, Georgios Mathioudakis, et al.

ICT IP Project

Deliverable D3.3

Integrated CHOReOS middleware - Enabling large-scale, QoS-aware adaptive choreographies

http://www.choreos.eu

Project Number	:	FP7-257178
Project Title	:	CHOReOS Large Scale Choreographies for the Future Internet

Deliverable Number	:	D3.3
Title of Deliverable	:	Integrated CHOReOS middleware - Enabling large-scale, QoS-aware adaptive choreographies
Nature of Deliverable	:	Report + Prototype
Dissemination level	:	Public
Licence	:	Creative Commons Attribution 3.0 License
Version	:	3.3
Contractual Delivery Date	:	30 September 2013
Actual Delivery Date	:	30 September 2013
Contributing WP	:	WP3
Editor(s)	:	Fabio Kon (USP)
Author(s)	:	Amira Ben Hamida (EBM), Fabio Kon (USP), Nelson Lago (USP), Apostolos Zarras (UOI), Dionysis Athanasopoulos (UOI), Dimitris Pilios (UOI), Panos Vassiliadis (UOI), Nikolaos Georgantas (Inria), Valerie Issarny (Inria), Georgios Mathioudakis (Inria), Georgios Bouloukakis (Inria), Yesid Jarma (Inria), Sara Hachem (Inria), Animesh Pathak (Inria)
Reviewer(s)	:	Daniel Batista (USP)

Abstract

This document describes the final implementation and the evaluation of the CHOReOS middleware. Evaluation is achieved both via the use of the middleware on CHOReOS use-cases and via synthetic experiments and simulation. The conclusion was that the implementation of the CHOReOS middleware has achieved a good level of maturity for an open source project and it is ready to be used in real-world, complex choreographies.

Keyword List

middleware, architecture, choreography, large-scale, SOA, Internet of Things, Cloud

Document History

Version	Changes	Author(s)
1.0	Initial Commits of the deliverable on SVN	Fabio Kon
1.1	Detailed descriptions of some components	Leonardo Leite
1.2	Detailed descriptions of all components	WP3 team
1.3	Added evaluation	WP3 team
1.4	General review	WP3 team
2.0	Final version after internal review	WP3 team
2.1	Final version after Valerie’s additional review	WP3 team

Document Reviews

Review	Date	Ver.	Reviewers	Comments
Outline	July 25th	0.1	Agreed outline by all partners	na
Draft	August 15th	1.0	Fabio Kon	Comments sent by mail
QA	October 9th	1.4	Daniel Batista	Comments sent by mail
PTC	October 18th	2.1	Valerie Issarny	Internal review and Valerie’s comments addressed and final version ready

Glossary, acronyms & abbreviations

Item	Description
API	Application Programming Interface
BC	Binding Component
CA	Consortium Agreement
CSDL	Client/Server Description Language
DL	Deliverable Leader
DoW	Description of Work
DPWSDL	DPWS Description Language
DSB	Distributed Service Bus
ESB	Enterprise Service Bus
GA	Generic Application
IAC	Industrial Advisory Committee
IOTS	Internet of Things Services
JBI	Java Business Integration
JMSDL	JMS Description Language
JSDL	JavaSpaces Description Language
LTS	Labeled Transition System
MST	Management Support Team
QoS	Quality of Service
OSS	Open Source Software
PL	Project Leader
PMT	Project Management Committee
PO	Project Officer
PTC	Project Technical Committee
PSDL	Publish/Subscribe Description Language
SL	Scientific Leader
SE	Service Engine
TDD	Test Driven Development
TSDL	Tuple Spaces Description Language
ULS	Ultra-Large Scale
WP	Work Package
WPL	Work Package Leader
xDL	Description Language
XSA	eXtensible Service Access
XSB	eXtensible Service Bus
XSC	eXecutable Service Composition
XSD	eXtensible Service Discovery

Table Of Contents

List Of Tables

List Of Figures

1 Introduction

The CHOReOS middleware is organized as a collection of software modules, which work together to support the development, deployment, and execution of complex, large-scale Web Service choreographies on a Future-Internet-like environment. It brings together technologies related to the Internet of Things, the Internet of Services, and Cloud Computing to facilitate such tasks. This is achieved by providing higher-level abstractions and services that organizations can rely upon to build complex, scalable, and adaptive service compositions.

This document describes the CHOReOS middleware implementation, whose open source code is available at `http://www.choreos.eu/bin/view/Download/Forge` and whose documentation is available at `http://www.choreos.eu/bin/view/Documentation/WebHome`. The document is organized as follows. Chapter 2 first presents the middleware overall architecture and then describes the implementation of each of its major components. Chapter 3 describes how to use the CHOReOS middleware as a choreography runtime environment. Chapter 4 presents our evaluation of the middleware, based both on the CHOReOS use cases and on synthetic experiments and simulations, stressing the middleware capabilities. Finally, we present our conclusions in Chapter 5.

2 CHOReOS Middleware: Architecture & Implementation

The CHOReOS middleware is organized in four major modules: eXecutable Service Composition (XSC), responsible for coordinating the composition of services and things, eXtensible Service Access (XSA), which provides the means to access services and things, eXtensible Service Discovery (XSD), which manages protocols and processes for discovery of services and things, and the Cloud & Grid Middleware, which manages computational resources and drives the deployment of choreographies.

Figure 2.1 depicts the overall architecture of the CHOReOS middleware, presenting each middleware module, the components inside them, and the interactions among the components. The figure also shows the interactions between the CHOReOS middleware developed with WP3 and the Governance and V&V Framework developed within WP4.

Application developers may choose to use all the middleware components to facilitate the construction of choreographies or, alternatively, to pick only the components that provide the functionality in which he/she might be interested. Details about the architecture and implementation of each module are described in this section. Details about how to use the middleware are provided in Section 3.

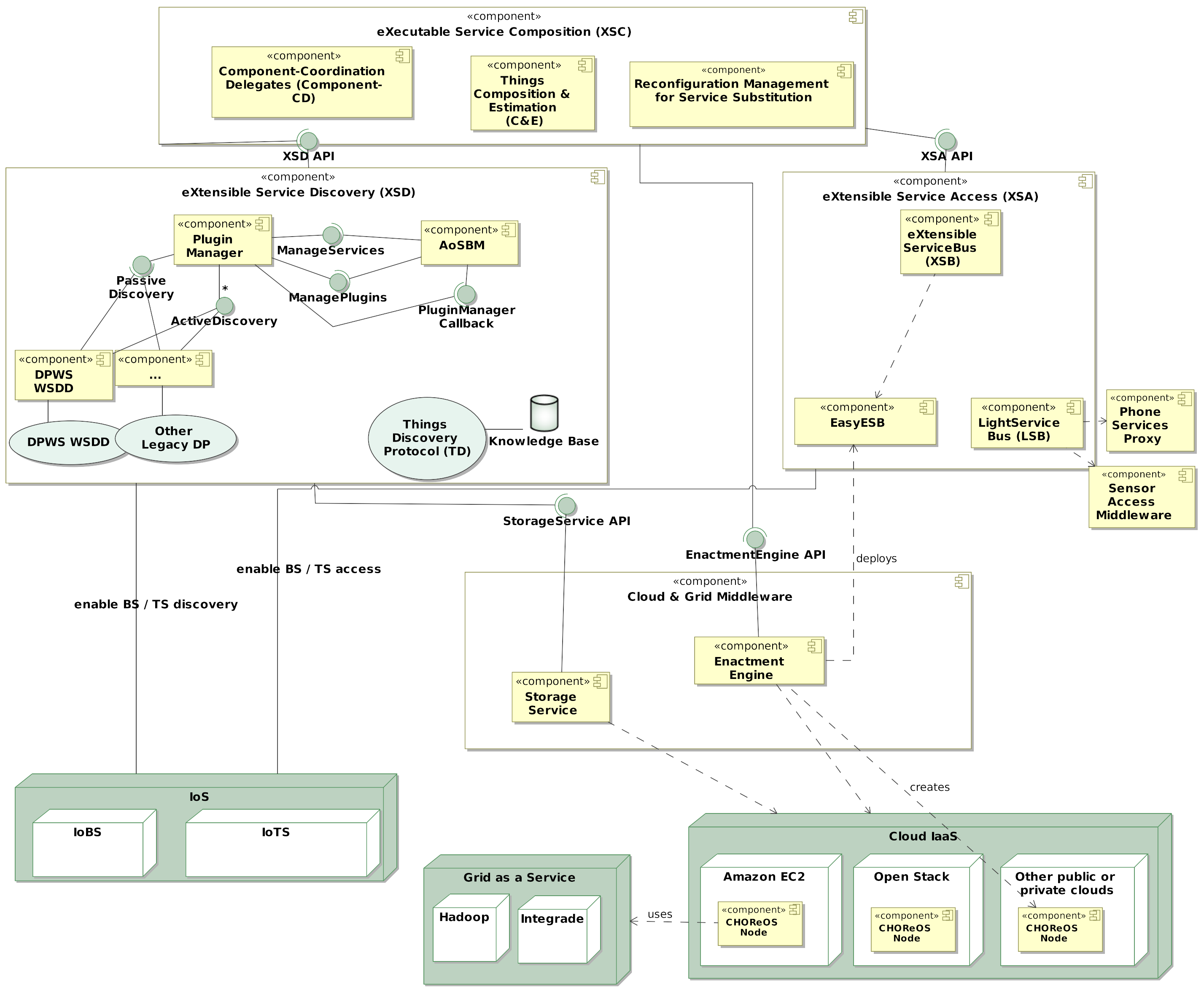


Figure 2.1: The CHOReOS middleware overall architecture

Core Contributions The eXecutable Service Composition (XSC) component provides facilities for the composition of both services and things. Services are composed by devising and deploying appropriate Coordination Delegate (CD) within the component-CD container. Basically, the component-CD container provides the communication primitives (e.g., UPDATE_STATE(), WAIT(), NOTIFY()) that are used by the coordination algorithm (detailed in deliverable D2.3 [15]) to exchange what we have called additional communication (i.e., coordination information). That is, CDs mediate (in a distributed way) the business-level interaction protocols of the participant services according to the coordination logic extracted from BPMN2 choreography diagrams by the synthesis processor (also detailed in deliverable D2.3 [15]). Things are composed by employing semantic composition specifications modeled as mathematical formulas: complex physical properties are derived from simpler ones provided by thing-based services. The XSC further comprises the abstraction-oriented service substitution facility, which allows to substitute services that provide similar functional properties. To this end, the services must be represented by the same abstraction [11] that provides the necessary mappings between the interfaces of the services.

The eXtensible Service Access (XSA) component provides facilities for interconnecting heterogeneous services and things. More specifically, the eXtended Service Bus (XSB) component enables interconnecting services that employ heterogeneous interaction paradigms, namely, the client-server (CS), publish-subscribe (PS) or tuple space (TS) paradigms. This is based on appropriate formal modeling and abstraction of each service middleware protocol into a connector representing the corresponding paradigm. Then, the CS, PS and TS connectors are further abstracted into a single higher-layer connector, the generic application (GA) connector, which unifies their features and enables their cross-connection. This formal approach is realized into the XSB service bus. XSB is implemented on top of the EasyESB component, which is a flexible, dynamic-topology enterprise service bus, also implemented within CHOReOS. XSB principles have also been applied to the Light Service Bus (LSB) component, which is a lightweight interconnection solution applying to things. LSB additionally supports data streaming protocols, which are particularly important in IoT.

The eXtensible Service Discovery (XSD) component provides facilities for the organization and the discovery of available services. To deal with scalability issues that concern the increasing amount of available services, the XSD employs the abstraction-oriented organization and discovery mechanisms (i.e., the mechanisms that constitute the AoSBM component) developed in WP2 [7]. Moreover, the XSD comprises a specialized Things Discovery protocol that leverages the concept of semantic abstractions for the organization of Thing-based services and offers facilities for the probabilistic registration and discovery of these services.

The Cloud & Grid Middleware provides a collection of Java classes and web services that make the process of deploying complex, large-scale choreographies composed of hundreds to thousands of services very easy. It also provides elasticity features to manage the number of replicas of specific services based on the workload (e.g., amount of user requests), resource utilization (e.g., usage of CPU or memory), and of QoS attributes (e.g., response time). To achieve that, it interacts with the CHOReOS Monitoring service [16] to detect when relevant events that should trigger dynamic reconfiguration occur.

We continue by describing in more detail the internal architecture of each of these middleware components.

2.1. eXecutable Service Composition

The eXecutable Service Composition (XSC) supports the composition of large number of services into business choreographies and large populations of things in response to user queries. Additionally, XSC supports service substitution for reconfiguration within choreographies. The following sections provide an overview of the corresponding XSC components, namely, Composition and Estimation (Section 2.1.1), and Reconfiguration Management for Service Substitution (Section 2.1.2).

2.1.1. Things Composition and Estimation C&E

The C&E component enables the composition of thing-based services. We exploit semantic technologies for the composition specifications. Given that thing-based services are sensing/actuating services in most cases, we model our compositions as mathematical formulas. For instance, wind chill can be estimated from temperature and wind speed measurements. Consequently, the composition specification is:

$$WC = (10\sqrt{V} - V + 10.5) \cdot (33 - T) \tag{2.1}$$

WC is the symbol for Windchill, V is the symbol of Windspeed, T is the symbol for Temperature. This information along with the mathematical formula are modeled in the IoT ontology. Concepts such as Temperature and Windspeed are referred to as `expansion concepts`.

Application developers can use this component to extract expansion concepts (e.g., Temperature and Windspeed), or compute the formulas to have an estimation of the initial concept value (e.g., Windchill). More precisely, the component extracts the expansion concepts and the formulas through SPARQL queries as strings. They are then parsed, automatically, into Java mathematical functions using the `exp4j`[1] jar. The component expects that services measuring the expansion concepts will be accessed for their measurements (using the QueryManager) and the formulas can then be computed using the acquired measurements as input. The details on the methods and the use of the component are presented in Section 3.3.

In our current implementation, the component is internally used by the QueryManager, to seamlessly execute compositions. However, we also provide it as a stand alone component in case the user wishes to exploit its functionalities separately.

2.1.2. AoSBM Service Substitution

Hereafter, we use the term AoSBM service substitution to refer to the realization of the Reconfiguration Management for Service Substitution component (Figure 2.1). The idea of service abstractions [9] has been further exploited to facilitate service substitution. The overall design of the *abstraction-oriented service substitution* approach has been specified in D3.2.2 [11], while the prototype implementation of the service substitution facility has been developed during the last year of the project. Conceptually the service substitution approach belongs to XSC. However, the implementation of the approach is coupled with the AoSBM facilities that extract service abstractions. Hence, the implementation of the approach is physically packaged in the AoSBM package (Section 2.3.1). The main concepts of the AoSBM service substitution approach are *functional abstraction services* and *impact analyzers*.

In particular, service substitution is performed between services that are represented by the same functional abstraction [9, 7]. Given a functional abstraction we generate a functional abstraction service that realizes the abstract interface, specified by the given functional abstraction [11]. The functional abstraction service is configured to delegate invocations, made to the abstract interface, to invocations on the interface of a concrete service that is hidden behind the functional abstraction service. To realize this delegation process, the functional abstraction service employs the mapping between the abstract interface and the interface of the hidden service instance, which is provided in the specification of the functional abstraction that represents the hidden service instance. The core of the delegation mechanism is a data translation module that transforms the inputs of the abstract invocation into corresponding inputs for the concrete invocation. Moreover, the data translation module transforms the outputs of the concrete invocation into corresponding outputs of the abstract invocation. The data translation module adapts the approach for the translation of XML documents, proposed in [24], to the case of Java objects. In particular, our approach assumes that the inputs/outputs are typical Java objects that conform to the JAX-WS[2] standard for XML-based Web services. The translation of a given Java object takes place in

[1]http://www.objecthunter.net/exp4j/

[2]http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index3.html

three phases[3]:

- First, the structure of the source object is recursively traversed using Java Reflection and the contents of the source object are transformed to a unified data representation that is employed by the translation mechanism. The unified representation is basically a relation that consists of a set of tuples.
- Second, the structure of the target object is recursively traversed using Java Reflection, and the unified data representation of the source object (produced in the first phase) is transformed according to the structure of the target object.
- Third, the target object is created using Java Reflection and it is filled with the contents of the source object that have been transformed according to the structure of the target object.

Substituting the hidden service instance with another service instance that is represented by the same functional abstraction amounts to change the mapping that is employed by the functional abstraction service for the delegation of the invocations. During the substitution, the service instances that depend on the substituted service instance are notified about the change. This notification is the main responsibility of the impact analyzer that is associated with the functional abstraction service. Upon the reception of such a notification, the dependent service instances are responsible for managing their local consistency.

2.2. eXtensible Service Access

The eXtensible Service Access (XSA) represents the main runtime infrastructure for accessing both services and things. In this section, we present the eXtensible Service Bus (XSB) as well as its specific paradigms. We dedicate Section 2.2.1 to recall the main concepts of the XSB. Then in Section 2.2.2, we briefly describe EasyESB, which offers the baseline middleware for the XSA framework. Moreover, it provides a suitable and scalable environment for business services to get involved into large choreographies. Finally, we describe in Section 2.2.3 the Light Service Bus (LSB). It is dedicated to Things-based services and addresses Internet of Things challenges. The several components of the XSA middleware are fully integrated in the CHOReOS IDRE.

2.2.1. XSB

The eXtensible Service Bus (XSB) provides support for seamless integration of heterogeneous interaction paradigms. With regard to middleware-supported interaction, the client-service (CS), publish-subscribe (PS), and tuple space (TS) paradigms are among the most widely employed ones, with numerous related middleware platforms, such as SOAP, REST, and Java RMI for client-server; JMS and SIENA for publish-subscribe; and JavaSpaces and LIME for Tuple space. Services relying on different interaction paradigms can be plugged into the CHOREOS XSB by employing binding components (BCs) that adapt between their native middleware and the common bus protocol. This adaptation is based on the abstractions, and in particular on the conversion between the native middleware, the corresponding CS/PS/TS abstraction, and the Generic Application (GA) abstraction, as depicted in Figure 2.2.

eXtensible Service Bus binding components (XSB BCs) are designed to be easily extensible to support new middleware platforms or new interaction paradigms. Hence, each subcomponent of an XSB BC is designed with 3 levels, the higher one being the most generic and that can be further refined into the two lower layers:

[3]We use the term *source object* to refer to the object that is transformed by the data translation module; we use the term target object to refer to the result of the transformation

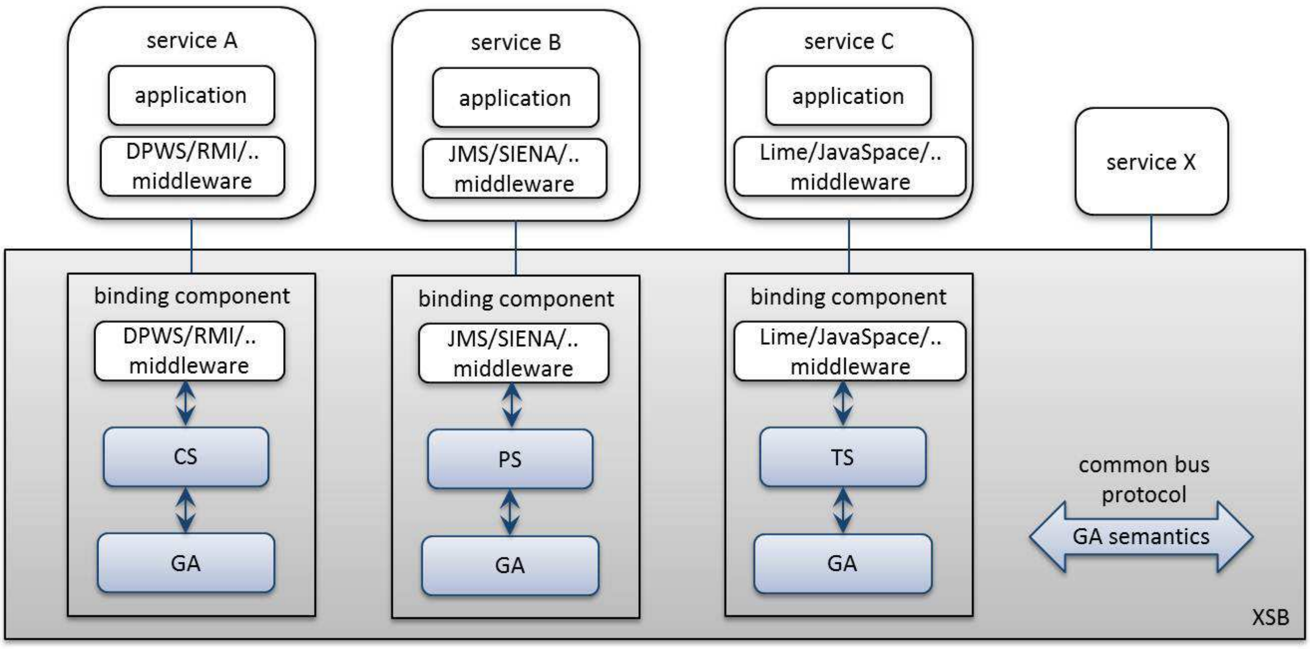


Figure 2.2: The CHOReOS eXtensible Service Bus

- The generic level provides APIs and functionalities that are shared among supporting implementations of all interaction paradigms.
- The interaction paradigm level specializes the APIs and functionalities of the previous level for the CS, PS and TS interaction paradigms.
- The middleware platform level specializes the APIs and functionalities of the previous level for a concrete middleware.

Besides a connector's API, we introduce an abstract interface description language (IDL) for specifying the open interfaces of systems that rely on middleware represented by the specific connector. Our IDLs are largely inspired from WSDL. We specify the IDLs conceptually, while we have also implemented each one of them as an XML schema document. Based on the flexibility of XML schema, an IDL can be easily refined in order to enable the description of a concrete system that is based on the connector, e.g., we can refine the abstract XML elements into the precise data structures and types of the specific middleware and system.

The architecture of an XSB Binding Component as provided by our architectural framework is depicted in Figure 2.3, where the main components are the xDL Processor, Core Engine, and Envelope for Substrate Bus.

xDL Processor

The *xDL processor* processes the descriptions of the services deployed on the XSB. It performs both parsing of paradigm DL descriptions (CSDL, PSDL, TSDL) and mapping of them to Generic Application descriptions (GADL), where the latter relies on XSLT-based transformations. At the generic and interaction paradigm level, these functions apply to abstract descriptions, while they become concrete at the middleware platform level. We use the XML schema extensibility mechanisms to specialize these functions from one architectural level to another. In particular, mapping paradigm DL to GADL descriptions further requires information about the semantic mapping between the constituent services that are integrated into the global application.

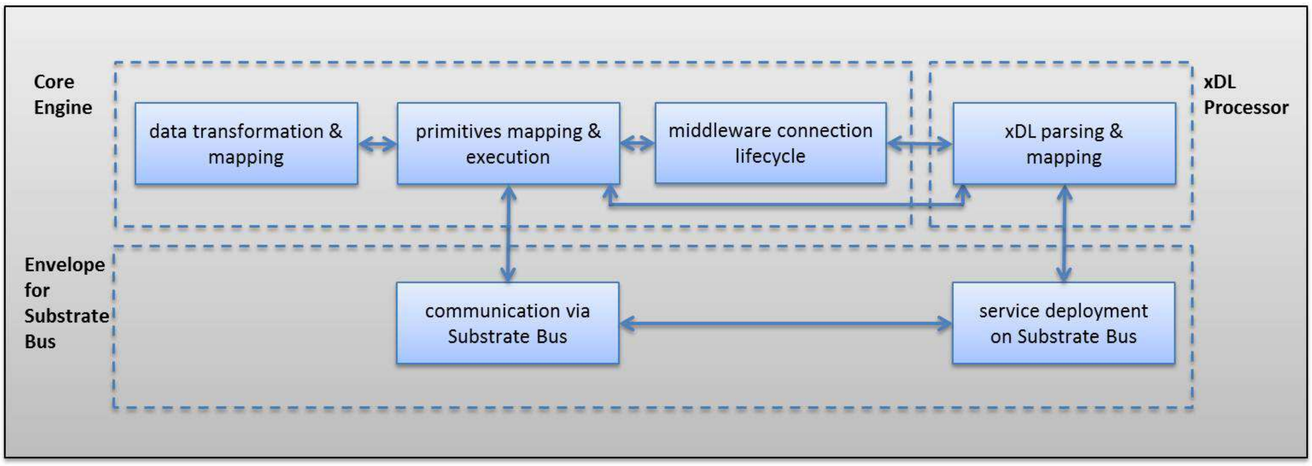

Core Engine
data transformation & mapping
primitives mapping & execution
middleware connection lifecycle
xDL parsing & mapping
xDL Processor
Envelope for Substrate Bus
communication via Substrate Bus
service deployment on Substrate Bus

Figure 2.3: Binding component architecture

Core Engine

The *Core Engine* provides all the mechanisms to:

1) transform service data to paradigm XML data, then map them to GA data, and vice versa
2) transform service primitives to paradigm primitives, then map them to GA primitives, and vice versa
3) manage the connections to the service deployed on the bus.

The above mechanisms cooperate with each other, as well as with xDL processor for retrieving information about the service. The interaction paradigm level customizes Core Engine APIs and functionalities to the CS, PS and TS paradigms and maps them to GA APIs and functionalities. At the middleware platform level, the Core Engine performs the real interactions with the concrete middleware to establish connections, as well as to send and receive data. Same as for DL descriptions, mapping paradigm data to GA data further requires information about the semantic mapping between the constituent services that form the global application.

Envelope for Substrate Bus

The *Envelope for Substrate Bus* makes the binding components (BCs) deployable on top of different communication substrates. It provides the mechanisms to:

1) extract/encapsulate GA primitives from/into bus connections, before/after passing/getting them to/from the Core Engine
2) transform service primitives to paradigm primitives, then map them to GA primitives, and vice versa
3) manage the lifecycle of a service deployed on XSB, after retrieving information about this service from the xDL processor.

These mechanisms are refined to support a new communication bus.

2.2.2. EasyESB

The extensible Service Access (XSA) middleware relies on the EasyESB Enterprise Service Bus (ESB) for integrating business services. EasyESB is an open source project licensed under the

LGPL. EasyESB adopts the Model-Driven Architecture (MDA) philosophy and is available for download at `http://research.petalslink.org/display/easyesb/EASYESB+-+Open+source+Lightweight+Services+Bus`. It is built on top of a modular and flexible topology that eases the creation of new bus nodes, their update, and removal. This ability makes possible the deployment of the bus nodes on the cloud infrastructure and their adaptation in case the cloud changes, e.g., when a new physical machine is added and, thus, a new EasyESB node is created and automatically linked to the previous nodes. The creation of new nodes allows the knowledge of all the services deployed on the neighboring nodes. Within the CHOReOS project, we benefit from this capability to deploy the EasyESB nodes on top of the Cloud infrastructure described in Section 2.4. EasyESB provides the backbone environment for the XSC middleware, the component-CD, by offering the needed multi-paradigm communication mechanisms. Finally, EasyESB eases the monitoring tasks implemented within WP4 by providing interceptors for the services bound to the bus. We implement an event-based mechanism in order to receive information captured by the interceptors. The generated information reports relevant data about the behavior of the services such as the response time, the payload, etc.

During the CHOReOS project, we have been improving EasyESB by several implementations in order to make it cope with the raised challenging objectives. For instance, we benefit from the cloud-aware infrastructure described in Section 2.4 and leverage the capabilities of the bus to be deployed on a scalable infrastructure. Joining both technologies represents a major step forward to the project as it enables accessing, integrating, choreographing and monitoring services from all over the world.

2.2.3. LSB

The goal of the CHOReOS Light Service Bus (LSB) is to enable access to data sensed by smart things using thing-based services. LSB comprises the *Sensor Access Middleware*, *Phone Services Proxy* and *LSB Binding Components*, which are described in the following.

Sensor Access Middleware

The Sensor Access Middleware is deployed on thing devices, in particular smartphones, and enables exposing their data (e.g., coming from the device sensors) as thing services. Figure 2.4 depicts the overall architecture of the Sensor Access Middleware, where the two main subcomponents are the *Thing Mediator* and the *Sensor Driver*. At the level of individual devices, a thing-based service using local sensors can utilize the Thing Mediator [8] for abstracting access to different sensors attached to the device. Individual vendors can contribute Sensor Drivers to their sensors that transparently bind with this mediator and provide access to sensed data through thing-based services. The Sensor Driver and Thing Mediator are described in more detail bellow:

Sensor Driver The Sensor Driver defines the API that should be implemented by the sensor driver developer. It consists of two main subcomponents: the `Sensor`, an Interface (API) for sensor drivers, which are the objects that hold the actual logic behind each sensor's operation and the `SensorInfo`, which contains metadata about a sensor.

Specifically, the functionalities that need to be implemented by each `Sensor` component are:

- exposure of its meta-description, including the name of the interface it implements and its unique (manufacturer-specific) device type
- configuration of the sensor parameters (setting and getting)
- access to the sensed data. This can be (i) *instantaneous*, when the query requests the latest sensing value of a Thing, (ii) *periodic*, which asynchronously returns at a constant rate (until canceled); and (iii) *event-based*, where a sensor replies whenever new data are available.

The functionalities that need to be implemented by each `SensorInfo` component include:

- providing the display name, description, and the name used for semantic matching of the sensor
- providing an instance of the sensor itself, thus acting as an instance of the *Factory* design pattern.

Thing Mediator The Thing Mediator component is meant to be accessed by the application developers and it consists of two subcomponents: the (i) `Mediator` and the (ii) `MediatorListener`.

The `Mediator` class provides the following functionality that can be used directly:

- discover all the sensors available on the device
- discover the instance of a specific sensor, specified by its interface
- trigger an immediate sensing by all sensors, where possible

Through the `Mediator`, the application can perform basic discovery and sensing actions with a single method call, abstracting away all the logical details such as creating separate processes in the OS, executing inter-process communication calls, synchronization, etc. All of these lower-level issues are all handled by the Mediator and the Daemon in an OS-independent manner. In addition, all sensors are treated homogeneously under a common API, whether they are microphones, thermometers, accelerometers, or cameras.

The `MediatorListener` is the abstract class that contains the functionalities to be implemented by the application developer to be able to retrieve the data from the sensors on the device. Specifically, the functionalities that the application developer should implement are:

- hooks to get notified when there is a change in the state of the mediator itself, or of the sensors attached to it
- hooks to get notified whenever there is new data available from one or more of the sensors.

Phone Services Proxy

In order to complete the LSB vision and enable access to thing-based services hosted by smart devices, the `Phone Services Proxy` component was introduced in the M24 release of the CHOReOS middleware as a solution to deal with a number of issues related to mobile communication, such as the use of NAT, transient IP addresses, enforcement of asymmetrical communication, and temporary disconnections. Clients (such as the `Things Query Manager`) that wish to communicate with thing-based services residing on mobile devices, send their requests and receive the corresponding results via the `Phone Services Proxy` (henceforth referred to as `proxy`), rather than directly to/from the devices themselves. Mobile devices retrieve, in a pull-based fashion, client requests that have been issued towards them via the `proxy` and, in turn, send replies to these requests in order for them to be forwarded to the respective clients. The introduced `proxy` mechanism as described here, constitutes one of the main building blocks towards the realization of the Light Service Bus (LSB), by providing a universal access solution to sensor data provided at the Things level.

Clients and mobile devices communicate with the `proxy` using RESTful interfaces, which are described in more detailed in Section 3.2.3.2. Service requests expire after a given time period (TTL) and, in case a timeout occurs, an appropriate error message is returned to the affected client. Moreover, each mobile device has a globally unique device ID that is used to address the specific device and, ultimately, its hosted services via the `proxy`. When mobile devices register their hosted services to the `Things Registry Manager`, they also provide a unique address for each of these services. This unique address includes (a) the address of the `proxy` associated with the specific mobile device (b) the unique deviceID, and (c) an identifier for the service registered.

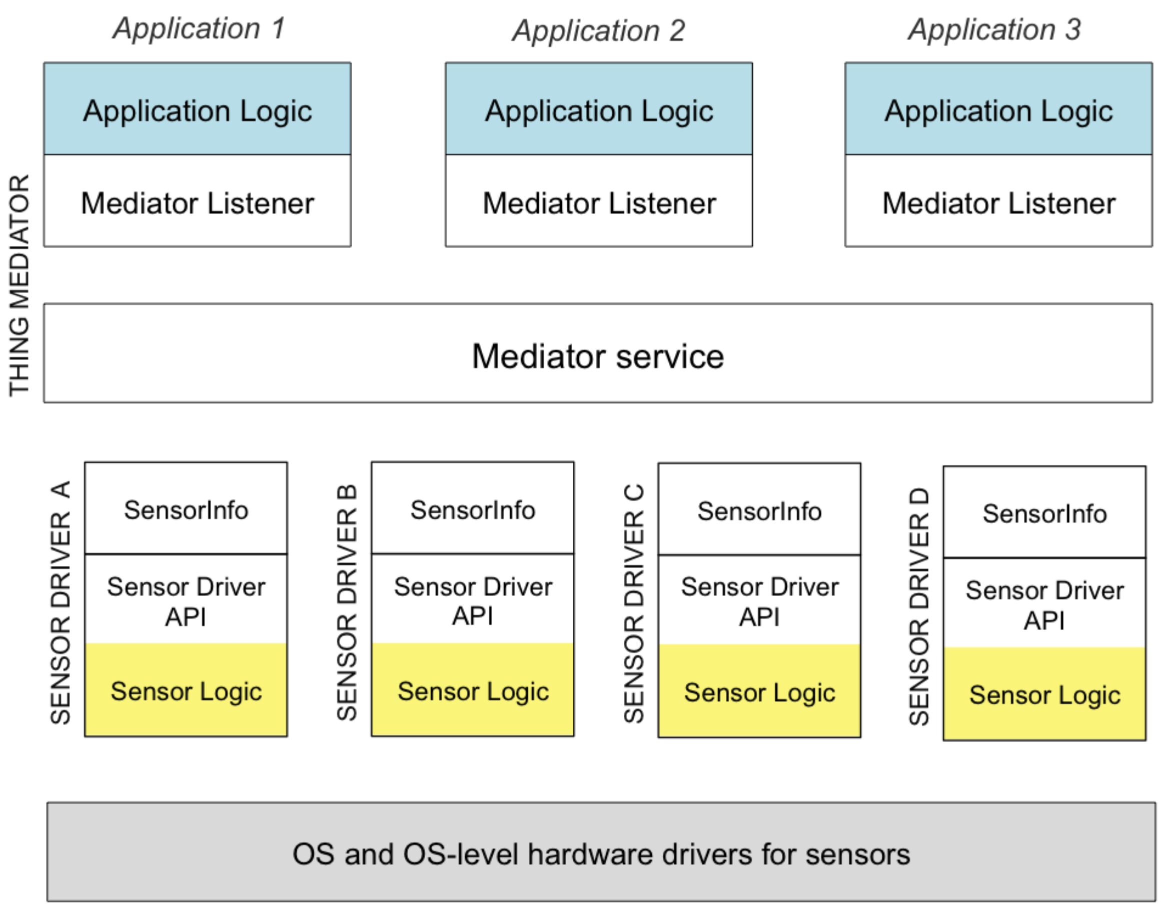


Figure 2.4: Sensor access middleware architecture

LSB Binding Components

As discussed in former CHOReOS deliverables, the major issue for service access towards dealing with the FI requirements is to be able to cope with the diversity of interaction protocols involved in IoBS and IoTS and, specifically, the integration and the pair-wide adaptation of these protocols. While the XSB, described in 2.2.1, addresses effectively heterogeneity and interoperability topics in the IoBS domain, a concrete access solution is required targeted particularly to the IoTS specifics, such as dynamic environments, resource constraints, data orientation, etc. Towards this, we introduce the binding components for LSB, being a lightweight version of the XSB for the IoTS domain. Like XSB, the LSB component is based on the CHOReOS connectors [9], which abstract the middleware-layer interaction protocols enabling Future Internet services to interact.

The design process of the LSB involved the identification of the interaction paradigms that are common in the IoT domain. Towards this end, an extensive survey was conducted on the state-of-the-art protocols being used in IoT-oriented solutions. The outcomes of this study confirmed the importance and wide use of the interaction paradigms already identified on XSB namely (i) Client-Server, (ii) Publish-Subscribe, and (iii) Tuple space, but also underline the existence of an additional paradigm focused on continuous interaction known as (iv) Streaming. However, Streaming can be considered as an extension to any of the interaction paradigms since, from a conceptual perspective, all of them must support both discrete and continuous protocols.

LSB binding components are implemented from scratch and are designed with focus on high-performance, small code footprint, compliance with all prominent IoT protocols, and easy-to-use and extend nature. Key point to the overall architectural design, as depicted in Figure 2.5, is the use of

REST as the common bus that handles the interactions between the several binding components and services. The REST approach provides the common integration infrastructure where all diverse interaction solutions can be plugged.

REST was selected among other WS-* solutions because of its simple architecture, lightweight nature, high performance in terms of message transmission speed and bandwidth consumption, and reliability, since it is based on HTTP. We chose the REST solution against DPWS, which was its main competitor, because REST provides uniform interfaces that makes its usage and debugging straightforward, a fact that made REST dominate the IoT market.

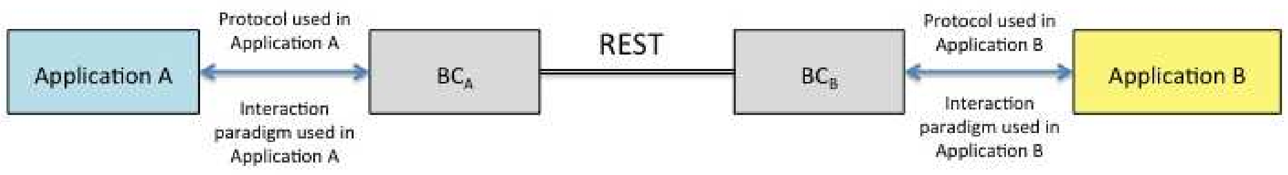

Application A
Protocol used in Application A
Interaction paradigm used in Application A
BC$_A$
REST
BC$_B$
Protocol used in Application B
Interaction paradigm used in Application B
Application B

Figure 2.5: LSB – overall architecture

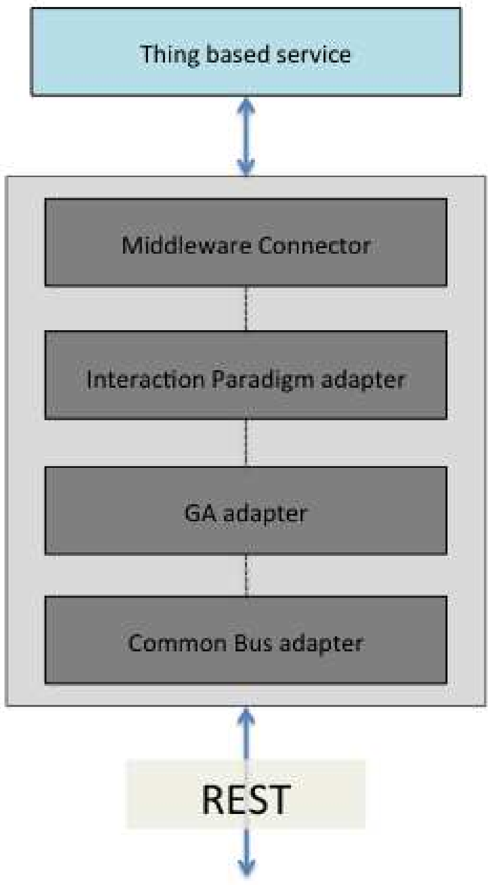


Figure 2.6: LSB – binding component architecture

A binding component of LSB consists of four layers/parts that are responsible for the transformation of the messages that run through the lightweight bus. These transformations are necessary to provide a cross-paradigm and cross-protocol communication solution. As it is depicted in Figure 2.6, the first layer represents the middleware connector of the service that initiates the interaction with the binding component and is responsible for maintaining contact with the service at any moment. The second layer involves the abstraction of the middleware-specific messages to the respective abstract primitives defined for each interaction paradigm, as described in D1.4 [9]. At the third layer of the architecture lies the GA adapter, a component that transforms the paradigm-based primitives to a more abstract type, that of GA. The GA's primitives are abstract enough to describe any type of message belonging to any of the interaction paradigms supported. The last layer of the binding component involves the mapping of a GA primitive to a specific call of the common bus, in this case, a REST call. All layers of the binding component support as well the inverse procedure of transformation e.g. from a REST call coming through the common bus to a GA primitive and finally to a middleware-specific call.

2.3. eXtensible Service Discovery

The eXtensible Service Discovery (XSD) supports the organization and discovery of both services and things among the vast corresponding populations available on the Future Internet. To this end, it includes a solution to service discovery based on service abstractions (Section 2.3.1), a solution to probabilistic things discovery (Section 2.3.2), and a plugin-based framework providing extensible support for any current or future discovery solution (Section 2.3.3).

2.3.1. AoSBM Discovery

The goal of the AoSBM component is to enable service lookup for business services in the context of the FI. To this end, we employ the concept of service abstractions, introduced in D1.3 [6] and refined in D1.4 [9]. The *modus operandi* of the main facilities of the AoSBM have been specified in D2.1 [7], D2.2 [10], and D2.3 [15], while the design of the AoSBM software itself was provided in D3.1 [8] and D3.2.2 [11]. Intuitively, a *functional abstraction* represents a group of services that offer similar functional properties; it is characterized by an *abstract interface* and a *mapping* between the abstract interface and the interfaces of the represented services. A *non-functional abstraction* represents a group of services that offer similar non-functional properties; it is characterized by an *abstract non-functional description* that consists of a set of layman's terms, one for each non-functional property of interest (e.g., response-time, availability) and a mapping *between* these terms to ranges of concrete values for each non-functional property of interest.

In the AoSBM approach, we exploit service abstractions as a means for organizing service descriptions in a registry and answering lookup queries for services. A lookup query is matched against service abstractions, instead of being matched against service descriptions of concrete services. Therefore, the query execution time scales up with the number of service abstractions, instead of scaling up with the number of available service descriptions. The service organization facility of the AoSBM accepts as input service descriptions that can be gathered from multiple sources (e.g., public registries, service portals) and produces hierarchically structured abstractions.

To enable the execution of lookup queries over abstraction-oriented organizations of service descriptions, we provide the Web Service Base Query Language (WSBQL). WSBQL is tailored to the concept of service abstractions and its syntax resembles the W3C XQuery standard. The WSBQL language comes along with a corresponding query engine which translates WSBQL queries to conventional SQL queries, which are issued to the AoSBM relational store [4]. Access to the query engine is provided via a dedicated REST API that has been developed during the last year of the project [15].

[4]More details concerning the WSBQL syntax and the query engine can be found at http://www.choreos.eu/bin/Documentation/Abstraction_Oriented_Service_Base_Management

2.3.2. Things Discovery

The Things Discovery component is aimed at handling the ultra-large scale of mobile things able to provide sensing/actuating tasks, mostly mobile phones. The thing-discovery protocol relies on probabilistic computations to control the participation of mobile things as needed.

In the registration phase, things hosting sensors/actuators should register their services through the `executeRegistrationQuery()` API. The discovery component then executes a set of probabilistic computations to determine whether or not the services are needed. The decision is based on the percentage of the area sensed by the already registered devices in the Registry Manager component. If the percentage is below an acceptable threshold then the new device should register its services and vice versa. The thresholds should be specified in the IoT ontologies we provide. Similarly, In the look-up phase, applications wishing to search for services need to access the methods that implement one of the three following APIs: `findServices()`, `findSubsetofServices()`, `findSubsetofServicesBasedOnCoverage()` . The first API allows the discovery component to return the *complete* set of devices that host services matching the request. The second and third APIs trigger the discovery component to perform probabilistic computations to determine a suitable *subset* of devices to access for the sensing/actuating services. More details on the use of the APIs are provided in Section 3.3

2.3.3. Plugin Manager

As specified in D3.1 [8], XSD relies on a plug-in based architecture, which enables incorporating support for any current or future service discovery solution applying to Business and Thing-based services. By this release, we are delivering the Plugin Manager component, the associated REST interfaces and the source code of abstract classes needed for plugin development. The Plugin Manager provides unified access to service discovery protocols present in the CHOReOS environment and essentially acts as an intermediate layer between them and the Abstraction-oriented Service Base. Via the Plugin Manager, the AoSBM gets populated with services discovered by these heterogeneous service discovery protocols. The Plugin Manager interfaces with these protocols via the appropriate plugins. Individual discovery protocols can be integrated in the XSD runtime system by extending the abstract *AbstractPlugin* class. Currently, the Java Reflection API is used for dynamic plugin loading. The Plugin Manager architecture was based on previous work by Inria (CONNECT Discovery Enabler [18]) and was extended by VTrip.

2.4. Cloud and Grid Middleware

The Cloud and Grid middleware addresses the scalability and distribution requirements identified in the CHOReOS project by offering three components:

- The Storage Service
- The Grid Service
- The Enactment Engine

The Storage Service simply automates the deployment of a database server on the cloud, providing its client with the credentials to access it. The current implementation deploys a traditional RDBMS, MySQL, but it is possible to extend it to handle other kinds of databases, including noSQL databases over a set of machines for load distribution. The Storage Service is actually an extension to the Enactment Engine, since most of it is implemented by using Enactment Engine components.

The Grid service is tailored at Grid Computing, which is offered as a service to developers needing to perform high-performance computing (HPC) for their applications. In other words, CHOReOS provides

Grid Computing as a Service, for applications with intensive computational requirements that could be not efficiently handled by a single machine hosting the service, such as a service that needs to process millions of images per day. According to our experience in the CHOReOS project within the past years, we anticipate that the Cloud Computing support offered by the CHOReOS middleware will be extremely useful for a wide variety of applications in multiple contexts. The Grid support, on the other hand, is a much more specialized service that would bring benefits to a more specific set of applications, e.g. that require Big Data Analytics. Specifically in the CHOReOS project, the three final use cases demonstrated interested in using the Cloud middleware, but not the Grid middleware[5]. Thus, although we provide a working implementation of both approaches, our major focus during the evaluation and assessment was on the Cloud part, which has a greater potential for adoption and dissemination.

The Enactment Engine is the main component of the Cloud and Grid Middleware and the focus of this section. It provides a powerful and elastic platform of hardware resources for the deployment of (1) the CHOReOS middleware, (2) business services, and (3) coordination delegates. New virtual machines are created on-the-fly as needed for the enactment of choreographies, and additional virtual machines may be created or removed during run-time according to demand or more sophisticated QoS constraints, avoiding extensive upfront investments on hardware for organizations participating in the choreography. It enables a fully-automated process for choreography deployment, which is a key feature to support Future Internet scale systems, since manually deploying large-scale choreographies is not feasible. Instances of Cloud components deployed on different organizations are able to collaborate among them to make a distributed choreography deployment across these organizations.

The Cloud and Grid components export a set of easy-to-use RESTful services. The choice of REST as communication protocol and interface style is in line with the current move on the Web services world, where the large majority of new services are based on REST and not on SOAP. This is due mainly to the better performance, less overhead, more flexibility, and ease of use provided by RESTful services in contexts such as the services provided by the Cloud and Grid components.

2.4.1. The CHOReOS Enactment Engine

The Enactment Engine deals with heterogeneity by using extensibility: although the current version can deploy only SOAP services packaged as JAR or WAR files, Enactment Engine users may extend it by writing a few new classes and configuration files to support new types of services. The Enactment Engine has also an extension point to allow the creation of new node allocation policies. Regarding the extensibility to support new Cloud IaaS providers (currently only Amazon EC2 and OpenStack are supported), all the developer must do is to write a Java class implementing the CloudProvider interface from our framework. The implementations of this interface for OpenStack and Amazon EC2 have around 200 lines of code; the size tends to be similar for other IaaS implementations.

Enactment Engine expects and handles faults of third-party components, which is important in large-scale distributed systems. An example of a typical failure in a cloud environment involves the virtual machine provisioning. When a new node is requested to the infrastructure provider, there is a chance that provisioning will fail. Moreover, some nodes may take much longer than average to be ready. In experiments conducted by us using the Amazon EC2 service, we verified that failures and long provisioning times affected up to 7% of the requests. Enactment Engine has a particular strategy to handle failures when invoking the IaaS provider (asking for VMs creation), that is called the *reserve of idle nodes*. When a request arrives, the Enactment Engine tries to create a new node. If the creation fails, or takes too long, an already created node is retrieved from a reserve of idle nodes. It avoids waiting again for another node creation. The reserve size is defined by configuration and it is refilled every time a reserve node is requested. The reserve approach has the extra cost to keep some nodes running in an idle state. However, only a small amount of extra nodes is necessary, since they will be used only when there is some failure on the cloud provider.

[5] In the beginning of the CHOReOS project, one of the use cases, Citizen journalism, planned to use the support from the Grid middleware, but the use case was dropped from the project and replaced by a new one.

Figure 2.7 illustrates the components that are necessary to enact a choreography. The Cloud Gateway and the Chef Solo are third-party software used by the Enactment Engine, whereas the Deployment Manager and the Choreography Deployer are components provided by the Enactment Engine.

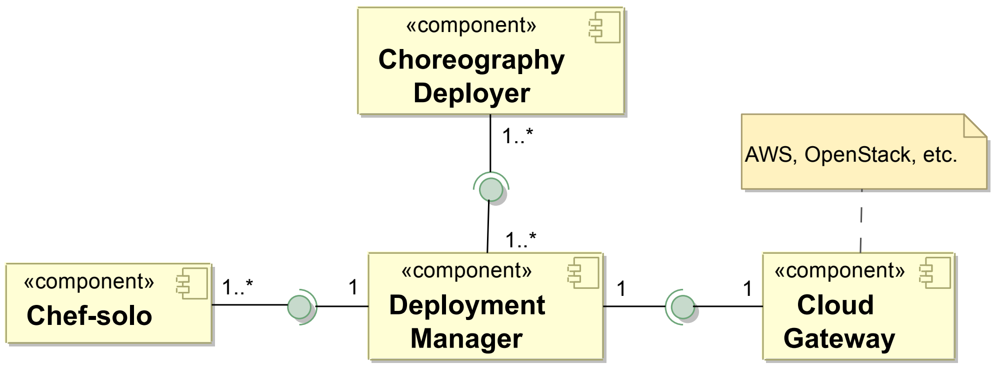


Figure 2.7: Enactment Engine components

Cloud Gateway creates and destroys virtual machines (also called *nodes*) in a cloud computing environment. This component is used by the Deployment Manager, which decides when to create or destroy the nodes. Currently, Amazon EC2 and OpenStack are supported as cloud gateways, but the Deployment Manager can be easily extended to support other platforms.

Chef Solo is a *configuration agent* installed in each node. It runs scripts for the installation of services and required middleware components. Such scripts are also called *Chef recipes* and are written in a Ruby-based domain-specific language.

Deployment Manager deploys services in a cloud environment. Through its *services API*, the Deployment Manager receives a declarative service specification and selects the node onto which the service will be deployed, possibly considering non-functional requirements of the service. The Deployment Manager converts the received specification to a script that implements the service preparation and launching processes. Using the *nodes API* provided by the Deployment Manager, one can request the upgrade of a node, which consists of running specific Chef Solo recipes on the specified node, and thus deploying services on the node.

Choreography Deployer exposes the *choreographies API* to provide support for the automated deployment of service choreographies or orchestrations (for our purposes, an orchestration here can be seen as equivalent to a simplified choreography in which coordination is centralized). The Choreography Deployer client must provide the choreography declarative specification, which contains the choreography architectural description and the locations of service packages. Based on this specification, the Choreography Deployer coordinates invocations to the multiple Deployment Managers belonging to the different participant organizations. When services are already running, the Choreography Deployer invokes consumer services, injecting on them the addresses of their dependencies.

In the next chapter, we describe in more detail how the various components of the CHOReOS middleware can be used by choreography developers to facilitate the coding and enactment of complex web service compositions.

3 How to Use the CHOReOS Middleware

The major goal of the CHOReOS middleware is to act as a choreography runtime. As a middleware, it acts as a software layer that sits between the underlying system (composed of OS, cloud, network, things, etc.) and the user-level application (e.g., a choreography). It raises the abstraction level provided to developers so that they can design, implement, and execute complex, large-scale service compositions with ease.

A developer may opt to use all of the CHOReOS middleware components as an integrated middleware infrastructure to support the system that he/she desires to build. Alternatively, he/she may pick and choose only the middleware components that provide the required services for a specific targeted application development. Table 3.1 lists the CHOReOS middleware components together with the requirements that they help to fulfill.

Middleware Component	Fulfilled requirement
Things C&E	Need for things composition, e.g., deriving complex physical properties from simpler ones provided by thing-based services
Service Substitution	Need for adaptation of the services used in a particular context
XSB	Need for seamless integration of heterogeneous interaction paradigms (client-service, publish-subscribe, and tuple space)
EasyESB	Need for integrating heterogeneous services Need for a runtime context enabling advanced SOA abilities such as composition monitoring, and governance
LSB	Need for accessing heterogeneous Things as services
Service Discovery	Sustain information about an ultra large amount of available services and facilitate efficient service lookup
Things Discovery	Need for efficiently handling the ultra-large population of mobile things able to provide a sought service
Grid as a Service	Need for high-performance computing, e.g., computationally-intensive algorithms or big-data applications
Enactment Engine	Need to deploy and reconfigure tens to thousands of services with geographically dispersed users, systems, and services while offering primitives for QoS management

Table 3.1: CHOReOS middleware components and associated requirements fulfilled

After deciding which CHOReOS middleware components to use, the development team will need to study the programmatic interfaces of each of the components and develop software that meets their specifications. A working installation of the CHOReOS middleware is also required; so, either the developer must have access to a previously configured CHOReOS installation (e.g., each company could maintain a CHOReOS installation that is shared among several projects) or they will need to download, build, and deploy the desired middleware components. Detailed development and installation guides for the middleware can be found at `http://www.choreos.eu/bin/view/Documentation/WebHome`. In the following sections of this chapter, we present an overview of how to use each of the major components.

The intended audience for this chapter are developers wishing to use the CHOReOS middleware to build their systems and applications, therefore, the remaining of this chapter contains low-level details of the installation and use of the middleware. Readers that are not interested in these specific details, may skip to Chapter 4 where the evaluation of the middleware is presented.

3.1. XSC – How to Use

3.1.1. Composition and Estimation – How to Use

The Composition and Estimation component is part of the XSC. It is specific to the thing-based service compositions. The source code of the project is available at `(root)/trunk/executable-service-composition/`. Developers wishing to use this component should download the source code and run the following command: `mvn clean install`. The command downloads the project's dependencies and use the resulting jar as a library in their code. Information on the technical requirements are provided in the install files available with the source code.

All compositions are specified semantically as mathematical formulas in the IoT ontologies, which are also available with the source code. This stems from the fact that all compositions are over sensing/actuating services that are tightly related to the real world and rules of mathematics and physics.

Developers are provided with two public APIs that allow them to request the alternatives to a missing service and compute the extracted formulas based on the measurements acquired from the alternative services. The APIs are presented below:

- `expandConcept()`: The method requires no input as it extracts the requested attribute from the request query. Using SPARQL queries the method extract the alternative concepts from the composition formulas and returns them to the user. Those concepts will them be send to the look-up component to find the service instances that measure the new concepts.

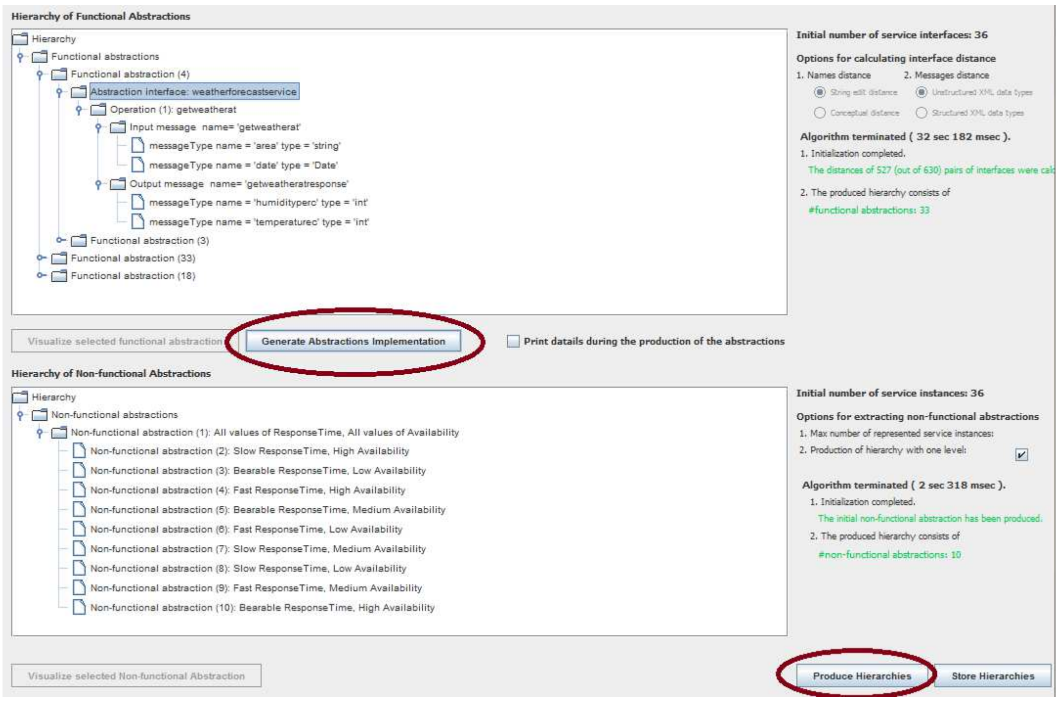
- `computeQueryConcept()`: The method requires no input as it uses the already extracted formulas from the call above. It acquires the measurements of the accessed services and sends them as input to the formulas. The result is then sent, internally to a fusion class for processing. The appropriate fusion/aggregation functions are specified in the IoT ontologies. The final result is returned to the user as a `SensorData` object that can be parsed into a `double`.

3.1.2. AoSBM Service Substitution – How to Use

The AoSBM service substitution facility can be used to generate automatically a functional abstraction service that realizes the abstract interface, specified by a given functional abstraction. This task is performed via the AoSBM GUI (Figure 3.1). Specifically, the AoSBM curator must create a functional abstraction that represents a set of similar services, using the AoSBM abstraction-oriented organization facility [15]. After this step, the generation of the functional abstraction service can take place. The generation process takes, as input, the AoSBM representation of the abstract interface that is specified by the functional abstraction and produces a JAX-WS-compliant Java interface that provides the operations, specified in the abstract interface. Listing 3.1 gives an example of the JAX-WS-compliant Java interface that is automatically generated for a functional abstraction that represents weather forecast services (more details regarding this example can be found in [11]).

```java
package org.ow2.choreos.abstractionGenerator.abstractionImpl.weatherforecastservice;
import org.ow2.choreos.abstractionGenerator.InputOutputDataTranslator;
import javax.jws.WebMethod;
import org.ow2.choreos.abstractionGenerator.Serialiazer;
import javax.jws.WebService;
import org.ow2.choreos.serviceRepresentation.functional.ServiceInterface;
import org.ow2.choreos.abstractionGenerator.BaseAbstractionImplService;
import org.ow2.choreos.abstractionGenerator.MappingsManipulation;
import org.ow2.choreos.abstractionGenerator.ClassCreation;

```


Visualize selected functional abstraction
Generate Abstractions Implementation
Visualize selected Non-functional Abstraction
Produce Hierarchies
Store Hierarchies

Hierarchy of Functional Abstractions

```
Hierarchy
  Functional abstractions
    Functional abstraction (4)
      Abstraction interface: weatherforecastservice
        Operation (1): getweatherat
          Input message  name= 'getweatherat'
            messageType name = 'area' type = 'string'
            messageType name = 'date' type = 'Date'
          Output message  name= 'getweatheratresponse'
            messageType name = 'humidityperc' type = 'int'
            messageType name = 'temperaturec' type = 'int'
      Functional abstraction (3)
    Functional abstraction (33)
    Functional abstraction (18)
```

[ ] Print datails during the production of the abstractions

Initial number of service interfaces: 36

Options for calculating interface distance

1. Names distance: [✓ String edit distance] Conceptual distance
2. Messages distance: [✓ Unstructured XML data types] Structured XML data types

Algorithm terminated (32 sec 182 msec).

1. Initialization completed.
   The distances of 527 (out of 630) pairs of interfaces were cal
2. The produced hierarchy consists of
   #functional abstractions: 33

Hierarchy of Non-functional Abstractions

```
Hierarchy
  Non-functional abstractions
    Non-functional abstraction (1): All values of ResponseTime, All values of Availability
      Non-functional abstraction (2): Slow ResponseTime, High Availability
      Non-functional abstraction (3): Bearable ResponseTime, Low Availability
      Non-functional abstraction (4): Fast ResponseTime, High Availability
      Non-functional abstraction (5): Bearable ResponseTime, Medium Availability
      Non-functional abstraction (6): Fast ResponseTime, Low Availability
      Non-functional abstraction (7): Slow ResponseTime, Medium Availability
      Non-functional abstraction (8): Slow ResponseTime, Low Availability
      Non-functional abstraction (9): Fast ResponseTime, Medium Availability
      Non-functional abstraction (10): Bearable ResponseTime, High Availability
```

Initial number of service instances: 36

Options for extracting non-functional abstractions

1. Max number of represented service instances:
2. Production of hierarchy with one level: [✓]

Algorithm terminated (2 sec 318 msec).

1. Initialization completed.
   The initial non-functional abstraction has been produced.
2. The produced hierarchy consists of
   #non-functional abstractions: 10

Figure 3.1: AoSBM graphical user interface

```java
@WebService
public interface WeatherForecastServiceInterface{

  @WebMethod public GetDailyWeatherForecastResponseOut getDailyWeatherForecast(GetDailyWeatherForecastIn input);

  @WebMethod public GetDailyWeatherForecastOnUpdateResponseOut getDailyWeatherForecastOnUpdate (GetDailyWeatherForecastOnUpdateIn input);

}
```

Listing 3.1: JAX-WS-compliant Java interface for a functional abstraction that represents weather forecast services

The AoSBM service substitution facility further generates the Java implementation of the JAX-WS-compliant Java interface. As shown in Listing 3.2, the Java implementation of the JAX-WS-compliant Java interface provides the implementation for each operation, specified in the abstract interface. The interface implementation extends the `BaseAbstractionImplService` class, which declares certain attributes that are used for the delegation of invocations that are made to the functional abstraction service to the concrete service that is hidden behind the functional abstraction service. Specifically, these attributes include the endpoint address of the hidden concrete service, and the mapping from the interface of the functional abstraction service to the interface of the hidden concrete service.

```java
package org.ow2.choreos.abstractionGenerator.abstractionImpl.weatherforecastservice;
import org.ow2.choreos.abstractionGenerator.InputOutputDataTranslator;
import javax.jws.WebMethod;
import org.ow2.choreos.abstractionGenerator.Serialiazer;
import javax.jws.WebService;
import org.ow2.choreos.serviceRepresentation.functional.ServiceInterface;
import org.ow2.choreos.abstractionGenerator.BaseAbstractionImplService;
import org.ow2.choreos.abstractionGenerator.MappingsManipulation;
import org.ow2.choreos.abstractionGenerator.ClassCreation;

@WebService()
public class WeatherForecastServiceImpl extends BaseAbstractionImplService implements WeatherForecastServiceInterface{

  public WeatherForecastServiceImpl (){
    super(”WeatherForecastService”);
  }

  public GetDailyWeatherForecastResponseOut getDailyWeatherForecast(GetDailyWeatherForecastIn input){

    GetDailyWeatherForecastResponseOut result = new GetDailyWeatherForecastResponseOut();

    LoadVariables();

    ServiceInterface abstractionServiceInterface = interfaceMapping.getServiceInterface(abstractionName);

    Object output = invokeConcreteService(input,”getDailyWeatherForecast”, abstractionServiceInterface);
```

```

      setOutPut (output, abstractionServiceInterface, "getDailyWeatherForecast" , result);

      return result;

    }

    public GetDailyWeatherForecastOnUpdateResponseOut getDailyWeatherForecastOnUpdate(GetDailyWeatherForecastOnUpdateIn input){

      GetDailyWeatherForecastOnUpdateResponseOut result = new GetDailyWeatherForecastOnUpdateResponseOut();

      LoadVariables();

      ServiceInterface abstractionServiceInterface = interfaceMapping.getServiceInterface(abstractionName);

      Object output = invokeConcreteService(input,"getDailyWeatherForecastOnUpdate", abstractionServiceInterface);

      setOutPut (output, abstractionServiceInterface, "getDailyWeatherForecastOnUpdate" , result);

      return result;

    }
}
```

Listing 3.2: Implementation of the JAX-WS-compliant Java interface

The implementation of each operation follows the same pattern:

- It has, as a parameter, an input object that encapsulates the input data for the operation, as specified by the given functional abstraction, and returns, as a result, an output object that encapsulates the input data, as specified by the given functional abstraction.
- The operation loads the values of the attributes that are used for the delegation of invocations, from disk, by calling the `LoadVariables()` method, which is inherited from the `BaseAbstractionImplService` class; the values are stored in the form of a .ser file.
- Then, the operation of the concrete service is invoked, by calling `invokeConcreteService()`. This method is also inherited from the `BaseAbstractionImplService` class; its major responsibilities are (1) to translate the data of the input object to the input parameters that are actually used for the call to the concrete service and (2) to perform the actual call.
- Finally, the data that are returned from the call to the concrete service are translated, to produce the output object of the operation. This is done by calling the `setOutput()` method.
- As discussed in Section 2.1.2, the input/output data translation process is based on the mapping information that is specified by the given functional abstraction and it is done using the Java reflection framework.

To substitute the concrete service that is hidden behind the functional abstraction service, the entity that is in charge of the substitution must call the `adapt()` operation that is inherited from the `BaseAbstractionImplService` class. The `adapt()` operation sets the interface name and the endpoint address of the service that is going to substitute the hidden concrete service. Moreover, it notifies the impact analyzers that have previously registered to the functional abstraction service [11]. A simpler alternative that does not involve the notification of impact analyzers amounts to calling the `setServiceInterface()` and the `setServiceInstance()` operations, which set, respectively, the name and the the endpoint address of the service that is going to substitute the hidden concrete service.

The generation of functional abstraction services makes use of the JAX-WS `wsimport` tool. Specifically, the tool is used for the creation of the JAX-WS stub classes, needed for the invocation of the concrete service that is hidden behind the functional abstraction service.

Detailed examples of functional abstraction services can be found in the CHOReOS AoSBM distribution (`(root)/trunk/extensible-service-discovery/AoSBM/src/main/resources/generatedAbstractionImplementations/`).

Moreover, examples of clients that use and adapt the generated functional abstraction services can be found in (`(root)/trunk/extensible-service-discovery/AoSBM/src/main/resources/clientForInvokingGeneratedAbstractions/`).

3.2. XSA – How to Use

3.2.1. XSB – How to Use

This section provides detailed instructions on setting-up and using the eXtensible Service Bus. Additional information are provided on-line available on: `http://choreos.eu/bin/Documentation/Extensible_Service_Bus` and in deliverable [14].

Technical Requirements To execute the XSB Binding Components with EasyESB the main requirements are:

1) Download Java JDK 1.6
2) Download maven 2.2.1
3) Download and unzip the Rio (http://www.rio-project.org) tarball

Setting-up the Development Environment The setup of the XSB Binding Components on a development environment like Eclipse, requires some steps. For example, the XSB DPWS Binding Component requires the following:

1) Download the sources of the corresponding B.C.:
   `connectors/clientServer/WebServices/DPWSBindingComponent/easyesbjmeds`
2) Add this repository to your `settings.xml` file. (Table 3.2):

```
    <repository>
      <id>choreos-petalsLink</id>
      <name>choreos maven repository</name>
      <url>http://maven.petalslink.com/repo</url>
</repository>
```

Table 3.2: Choreos maven repository

3) Execute `mvn install` to obtain the binding, or `mvn install -Pdistrib` to obtain a easyESB distribution with the DPWS Binding Component.
4) Run `target/easyesbjmeds/bin/startup.bat` to start the easyESB distribution with the DPWS Binding Component.

Using the Application You can verify the binding component by running two scenarios. In the first scenario, two DPWS applications have a one-way exchange through the XSB Binding Components. The first application sends a notification that is received by the second application as a one-way invocation. You can run the above scenario by following the following steps:

1) Start the application that receive the one-way invocation:
   `java -jar Easyesbjmeds-1.0-SNAPSHOT-OneWaySystem.jar`

2) In the menu, choose option 1 to start the system.

3) Start an easy ESB node with the DPWS Binding Component.

4) Deploy the corresponding service unit:
`src/test/resources/oneWay/UnitOneWay.xml`

5) Start a second easy ESB node with a DPWS Binding Component.

6) Deploy the service unit of the system that sends the notification:
`src/test/resources/notification/UnitNotification.xml`

7) Start the application that sends the notification:
`java -jar Easyesbjmeds-1.0-SNAPSHOT-NotificationSystem.jar`

3.2.2. EasyESB – How to Use

The EasyESB user manual is fully described in the deliverable D5.3.2 [14]. Additional relevant user guides and requirements of the EasyESB Bus are provided at `http://choreos.eu/bin/Documentation/easyesb`. The bus is open source, both source code and binaries are available. Both Linux and Windows distributions are provided. A list of specific user guides about administrating the bus, connecting it to new nodes, etc., is available at `https://research.petalslink.org/display/easyesb/Specific+How-tos`.

3.2.3. LSB – How to Use

Sensor Access Middleware – How to Use

This section describes the requirements of the Sensor Access Middleware component and provides detailed instructions on setting-up and using the project. Additional information are provided online and is available at `http://choreos.eu/bin/view/Documentation/Sensor_Access_Middleware`.

Technical Requirements The Sensor Access Middleware component is written in Java and implemented as an Android application. The Android operating system is chosen among others for its open-source nature and the flexibility it provides as a development platform. For running the application on a mobile device, the lowest supported Android version is the 2.3 Gingerbread. According to Google, over 97.6% of the enabled Android devices use an Android version higher than 2.3.

Setting-up the Development Environment The set-up of the application project in a development environment does not require any special procedure. The project just needs be imported to the active workspace, for *Eclipse* use File → Import → Existing Projects into Workspace. A JDK 1.6 or higher and the Android SDK are required to achieve a successful set-up.

Using the Application To run the application, a compatible Android device or an emulator is needed. However, since the Android emulator introduces some issues regarding sensors, the use of a real device is encouraged. Prior to starting the Things services on the Phone, the remote addresses (URLs) of the `Phone Services Proxy` and the `Registry Manager` should be configured through the settings window in the application. Communication with the proxy is required to guarantee the access to the Things services as discussed in Section 2.2.3.2. The Registry Manager is contacted by the application to execute the registering and un-registering procedures.

After starting the Things services (by clicking the enable button in the app), the onboard REST server starts automatically and exposes the defined API. This API is summarized on Table 3.3, where the IP address and port are the ones assigned to the device when connected to a local network:

URI	Method	Parameters
`http://128.0.0.192:8080/_servicename_`	GET	getnoiselevel, getlocation, gettemperature, etc.

Table 3.3: Sensor access middleware REST API

Responses to sensing requests are sent back to requesters as JSON messages. Each response message contains (i) a timestamp defining the time of the sensing, (ii) the sensing data type, e.g., the CHOReOS custom datatype of noise level, and (iii) the value itself. An example response is shown in Table 3.4.

Response JSON message

```
{
  "sensorDataResponse":{
    "timestamp":1378975233604,
    "value":"75.878",
    "dataType":"org.ow2.choreos.sensordata.double.noiselevel"
  }
}
```

Table 3.4: Response containing the sensing value

Phone Services Proxy – How to Use

As described in Section 2.2.3.2, the `Phone Services Proxy` is designed to provide a robust and complete access solution to remote Thing services. Therefore, by definition, it acts as an intermediate between the devices providing the services and the requesters, being clients or applications. To guarantee the flexibility and interoperability of these communications the proxy is based on a defined REST API. Additional information are provided online and is available at `http://choreos.eu/bin/view/Documentation/Phone_Services_Proxy`.

Technical Requirements The Phone Services Proxy is a Web application project written in Java. It is packaged as a war file and can be deployed on a `Tomcat` server (version 6 or higher).

Setting-up the Development Environment The proxy server is developed based on the `Maven` project management tool. Hence, the setup of the web application on a development environment like Eclipse, requires a working installation of `Maven` (version 2 or higher). After the typical importing procedure to the workspace of the environment, the execution of the `mvn clean install` command is required for the download of the dependencies and the building of the project.

Using the Proxy After the deployment of the Proxy server, it is ready to start serving requests for Things services access. A requester uses the corresponding URI defined in the API (Table 3.5) to issue a request. Included in this request are the device's unique ID and the name of the service (e.g., getnoiselevel). A pending requests data structure inside the proxy is used for keeping the requests for all devices and their services.

Each device that supports the `Proxy` contacts it periodically and checks if any new requests are available concerning any of the Thing services offered. This is done with the use of the `pendingrequest`

URI	Method
http://things.inria.fr/proxy/_deviceID_/_servicename_	GET
http://things.inria.fr/proxy/pendingrequests/_deviceID_	GET
http://things.inria.fr/proxy/sendresponse/_deviceID_/_servicename_	POST

Table 3.5: Phone services proxy REST API

method of the proxy's defined API. When the `pendingrequests` call is received on the proxy, the respective data structure is checked and, if any requested services are found for this device, a response is constructed containing a JSON message that includes them. An example of such a response is shown in Table 3.6.

Response for pending requests
`{ "pendingRequests":[ { "serviceName":"getnoiselevel" } ] }`

```
{
   "pendingRequests":[
      {
         "serviceName":"getnoiselevel"
      }
   ]
}
```

Table 3.6: Response for pending requests

In the case that any services are requested from a device, a sensing session starts in order to gather new data from the sensors. When ready, the data are sent back to the proxy using a POST call (sendresponse). The body of an example call of this type is shown in Table 3.4. The newly received data are forwarded back to the requester as a response to the initial request. If the service requested is unavailable or takes too long to respond, a timeout occurs on the requester.

LSB Binding Components – How to Use

An LSB binding component (BC) acts as an intermediate between the service it is binded to and the outside world. Any outgoing calls from the service will go through the binding component and the same occurs for incoming connections. By doing so, the messages are abstracted from the specific protocol and interaction paradigm to the generic type and then transmitted using a common bus, in this case REST. A BC can be attached to a service just by changing the addresses of the calls. The initial step is to configure the destination address on the service that initiates the call (e.g. Application A in Figure 2.5) and then the BC itself should be configured to include the final destination address of the messages. Additional information are provided online and is available at http://choreos.eu/bin/view/Documentation/Binding_Components_LSB.

Technical Requirements A binding component reserves two TCP ports (8090, 8092), used for receiving the incoming requests from the common bus and the service it is attached to.

Setting-up the Development Environment The project does not require special configuration in any development environment. After the typical import, the command `mvn clean install` should be executed to download the external libraries required. The binding components for LSB are written in Java, thus a working JVM is needed for the execution.

Extending the Binding Components As described in section 2.2.3.3 the first layer of a binding component for LSB is the middleware connector, which is responsible for handling the communication with the service connected. In its current implementation the binding components support some of the dominant protocols in the IoT domain, which are used to provide the proof of concept. However, a developer could easily extend the support of any type of service to the binding components. This can be done by configuring the middleware connector so as to map the outgoing requests and the incoming calls between the specific protocol of the service and the paradigm adapter layer. The rest of the procedure is handled automatically by LSB.

3.3. XSD – How to Use

3.3.1. AoSBM – How to Use

At a glance, the usage of the AoSBM concerns two key actors: (1) the AoSBM curator, who is responsible for the registration of information about available services and the organization of the services with respect to abstractions and (2) the AoSBM user, who uses the AoSBM to perform service lookup queries.

In the first place, the AoSBM curator is supposed to install and configure the AoSBM in one or more nodes. This task is fairly easy; detailed instructions can be found in the CHOReOS installation and user guides [14], in the CHOReOS courseware [17], and at the CHOReOS documentation Web site (`http://choreos.eu/bin/Documentation/Abstraction_Oriented_Service_Base_Management`). Following, the AoSBM curator must populate the AoSBM with information about available service descriptions and organize this information with respect to abstractions. The service registration and organization tasks can be performed via the AoSBM GUI; again detailed instructions concerning these tasks can be found in the CHOReOS installation and user guides [14], in the CHOReOS courseware [17], and at the CHOReOS documentation Web site.

Once the AoSBM is setup, there are two main usage options for the AoSBM users who wish to execute service lookup queries. The first option is to write down a WSBQL query, load the query via the AoSBM GUI and execute it. As before, these tasks are demonstrated in detail in the CHOReOS installation and user guides [14] and in the CHOReOS courseware [17]. Moreover, a document that refers to the syntax and semantics of WSBQL is available at the CHOReOS documentation Web site. The second option for using the AoSBM query facilities is via the `QueryEngineService` REST API that we developed for this specific purpose. The modus operandi of the API is explained in depth in [15]. Moreover, a Java doc that explains the functionalities that are provided can be found in the CHOReOS AoSBM distribution (`(root)/trunk/extensible-service-discovery/AoSBM/doc/`).

3.3.2. Things Discovery – How to Use

The Things Discovery component enables the registration and look up for thing-based services, mostly sensors/actuators services. The component is to be used by developers wishing to advertise thing-based services to the Registry Manager and developers wishing to provide applications that consume the registered services.

Users are required to integrate the TD components, namely the `RegistrationManager` and the `QueryManager` with their application's code. The source code is available at the following address: `(root)/trunk/extensible-service-discovery/`. For each of the projects, developers should run a `mvn clean install` to acquire all the project's dependencies. Developers can then add the resulting jar files as libraries in their code (More information is provided in the install files provided with the source code of the `RegistrationManager` and `QueryManager`). Finally, the address of the RegistryManager should be provided by the user.

Users who wish to advertise their services can do so through the `executeRegistrationQuery()` method by providing the following: the deviceId, the name of the service, the type of the service, the type of the sensor/actuator, the concept it measures/acts on, the type of the data the service produces, the current location of the device, its expected mobility path and the unit of the provided measurement.

We are currently working on decreasing the set of parameters by extracting some of them directly from the IoT ontologies. It is possible that the component declines the developer's request to register its services for a duration of time. This is due to our probabilistic computations that allow service instances to register if needed only. The decision is valid for the request time and location only. As such, the developer can retry at later periods. The method returns true if registration was successful and false otherwise.

Users developing applications that consume registered services have three alternatives to discovering those services:

- `findServices()` method. The method takes the attribute to measure/act upon and the location of interest as input. It returns the addresses of **all** services the satisfy the request.
- `findSubsetOfServices()` method. The method takes, as input, the attribute to measure/act upon, the location of interest and the desired spatial probability distribution of devices hosting the services and the number n of the required services. It returns the addresses of n services that satisfy the request. For the current version of the work, we support two distribution types: Normal distribution if the location of interest is a point in space (e.g., an intersection on the road), Uniform distribution if the location of interest is an area (e.g., Paris city).
- `findSubsetOfServicesBasedOnCoverage()` method. The method takes the attribute to measure/act upon, the location of interest and the spatial probabilisty distribution of devices hosting the services and the minimum required coverage percentage, i.e., the minimum acceptable percentage of the area to be covered/sensed by the selected services. It returns the addresses of a number of services, unknown a priori, that satisfy the request.

The Registry Manager is a Web Service that developers should deploy in order to be able to register their sensing/actuating services. To deploy and run the Registry Manager, users need to download the source code available at `(root)/trunk/extensible-service-discovery/registry_manager/` and run the `mvn clean install` command. This will generate a war file that should be deployed in a Tomcat server.

3.3.3. Plugin Manager and Plugins – How to Use

Plugins enable access to diverse service discovery protocols. All plugins must extend the *AbstractPlugin* class, thus defining a new plugin type. The Plugin Manager can create multiple instances of the same plugin type. For example, for N UDDI registries the Plugin Manager would create N instances of the "UDDI" plugin type. Each plugin instance has the responsibility to deliver service-related information from the underlying service registry to the Plugin Manager. Based on how the communication between the plugin and the service registry takes place, the plugins can be categorized by their supported discovery mode: *Active, Passive, or both*.

Plugins that operate in "passive discovery mode" are passive receivers of messages from the underlying service registry. These messages indicate that a service has been added/removed/updated in the registry. As soon as a registry message is received, it is up to the plugin to inform the Plugin Manager accordingly. In the event of a service addition or update, the plugin must also provide the respective service description in a unified format. This "unified format" is currently being defined.

On the other hand, plugins that operate in "active discovery mode" must explicitly request service descriptions from the underlying registry. This mode is needed for registries that do not provide an eventing mechanism. Plugins that support this mode implement the `retrieveServices()` method, which is invoked by the Plugin Manager. The `retrieveServices()` method call results in the (indirect and asynchronous) retrieval of all services that match a specific "ServiceFilter", which is passed as a parameter. The exact format of the "ServiceFilter" has yet to be formally defined and will probably take into account numerous parameters like service semantics, underlying transport protocol, request timeout, upper/lower limit to the number of services returned etc.

In addition, a plugin could support both modes (e.g. DPWS/WS-Discovery)

Finally, in order to evaluate the current Plugin Manager API, a plugin for a ”dummy” registry was developed. This ”dummy” registry acts as a daemon that reads service descriptions from a specific folder and informs its clients for new, updated and deleted services. Hence, the associated plugin operates in passive-discovery mode.

3.4. Cloud and Grid – How to Use

In this section, we focus on describing briefly the required steps to use the CHOReOS Cloud Enactment Engine to deploy a web service choreography. The use of the Grid middleware, as explained in Section 2.4, was not the focus of the last year of the CHOReOS project; readers interested in its details can refer to deliverable D3.2.1 and the online documentation at `http://choreos.eu/bin/view/Documentation/grid_doc`.

This section is not aimed to make a developer fully capable of using the Enactment Engine, but rather to provide an overall idea about the technical work necessary to setup and use the Enactment Engine. Detailed and comprehensive instructions are provided by the Enactment Engine User Guide, available at `http://choreos.eu/bin/Documentation/enactment_engine_doc`; for the interested reader, we highly recommend downloading the User Guide from this page.

In the current document, we call *deployer* the Enactment Engine user, i.e., the one who wants to deploy a choreography, and the *target environment* the machines where services are deployed. Such terminology is in accordance to the OMG’s component-based systems deployment standard [26]. Note that if you use CHOReOS full capabilities, the deployer may be the CHOReOS system itself.

After downloading and compiling the Enactment Engine source code, it is necessary to configure the Enactment Engine, providing it some information about the node allocation policy (how services must be distributed among different nodes) and the used cloud gateway. A cloud gateway here is just any service capable of providing new virtual machines on demand. In this step, one can enable or not some extra features of the Enactment Engine, such as automatically destroying inactive nodes, monitoring nodes with Ganglia, monitoring services with the EasyESB, linking an EasierBSM instance to the EasyESB nodes to be created, and so on. All of this configuration is performed by editing text properties files.

Once the Enactment Engine is running, the deployer must prepare the Enactment Engine input. Such input describes all the necessary information for choreography deployment, and it encompass mainly: in which kind of package each service is distributed (JAR, WAR, etc.), the package URL, which services depend on which services within the choreography, and in which cloud each service must be deployed. This description must adhere to the choreography specification data model defined by the Enactment Engine (Figure 3.2); it can be eother provided in XML format through the Enactment Engine REST API, or one can invoke the Enactment Engine using the Java API and build the choreography specification using Java objects.

After the choreography specification is ready, the deployer must invoke the Enactment Engine by using the REST API or the Java API[1]. The Enactment Engine will then deploy the services on the target environment and bind the services, so they can call each other. Service binding is performed by invoking the `setInvocationAddress` operation provide by services. After the deployment is completed, the Enactment Engine returns a choreography description (Figure 3.2), which specifies in which node each service was deployed and how one can access each one of the deployed services. This response is also returned in the XML format or as Java objects, depending on the deployer’s choice.

While the deployed choreography is active, it is possible to perform modifications to it. For instance, one may decide to switch from using a service offered by one provider to a compatible service by a different provider, or to increase/decrease the number of deployed replicas of a given service in order

[1]The Java API actually uses the REST API under the hood.

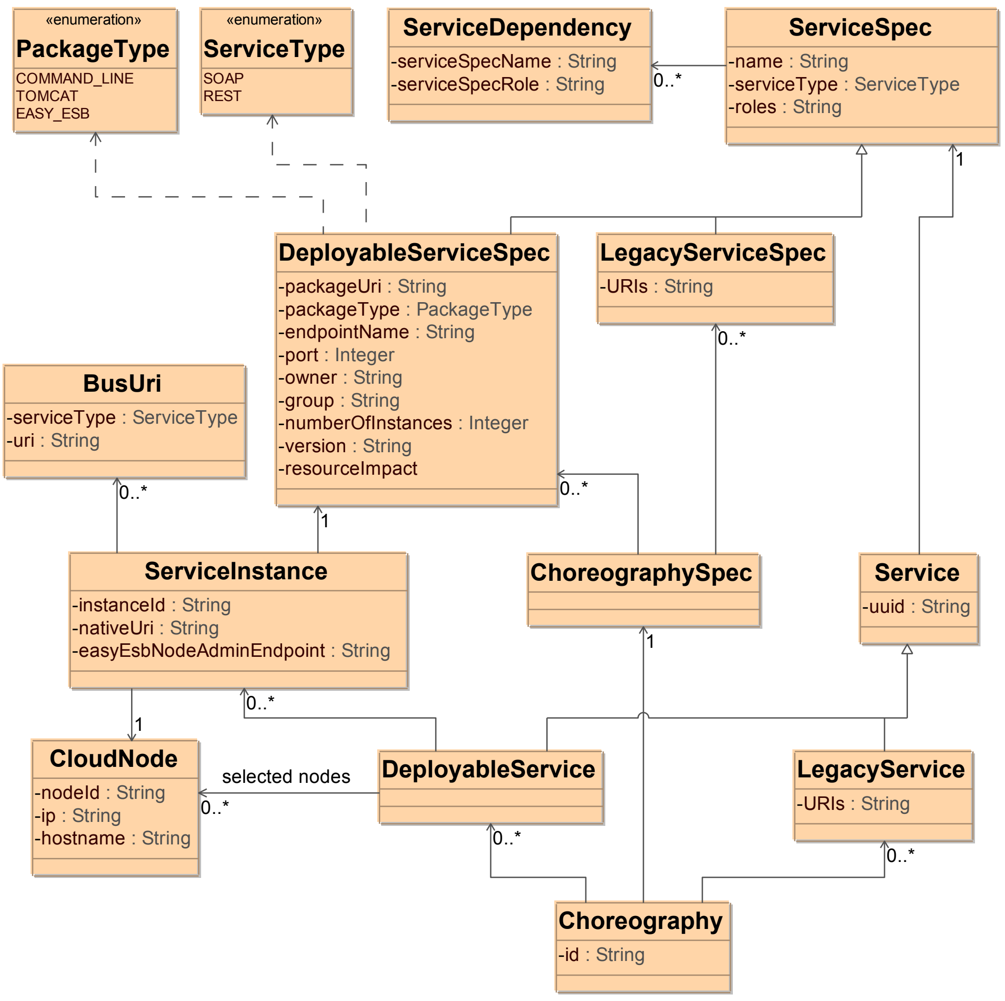


Figure 3.2: The Enactment Engine data model for choreography specification

to adapt to fluctuations in usage load. The deployer simply uses the API again to submit an updated version of the choreography specification to the Enactment Engine and requests the reenactment of the choreography. The Enactment Engine, in turn, detects the modifications made to the choreography and performs the requested modifications, by deploying new versions of services, removing service replicas etc.

This capability, together with the flexibility offered by the CHOReOS monitoring subsystem, presents the user with the framework necessary to adjust the run-time environment of the choreography according to QoS parameters and constraints, such as response time or cost. The monitoring subsystem is able to collect low-level data from virtual machines, such as disk and network bandwidth utilization or memory and cpu consumption, as well as higher-level data from the EasyESB bus, specifically several aspects related to service response time. This collected data is forwarded to the Glimpse Complex Event Processor [12, 13] for further processing. The CEP is a rule-based system which allows the user to define, at run-time, the event detection rules he is interested in together with the action to be taken whenever one such rule is triggered. These rules not only are able to identify QoS violations but also may correlate them with the collected low-level data, empowering the user to create different specific actions to be taken according to the context of each QoS violation. Such actions tipically consist of requests for the Enactment Engine to modify the running choreography (add or remove service replicas, migrate a service to a more or less powerful virtual node, etc.) to maintain the required QoS. An example is discussed in the already mentioned Enactment Engine User Guide.

For each kind of service package, there are some rules to be followed. For example, JAR files must be runnable JARs, WAR files providing SOAP services must contain the sun-jaxws.xml file, and so on. Service implementation should also adhere to some minor requirements, such as logging on files, rather than on console, not relying on absolute paths of resources, not depending on run-time objects living on other services, and so on. Moreover, each service must properly implement the `setInvocationAddress` operation, so it can receive the addresses of its dependencies, that are other services within the choreography. The complete set of requirements and guidelines for using the Enactment Engine are available at http://choreos.eu/bin/Documentation/enactment_engine_doc.

4 Evaluation

The evaluation of middleware components was based not only on use cases, but also on experiments and simulations. Note that when using the middleware components in the use cases, but also for experimental evaluation purposes, certain components are employed together as a group across the major modules (XSC, XSA, XSD, Cloud & Grid) of the CHOReOS middleware architecture. Thus, in the following sections: AoSBM Discovery and AoSBM Service Substitution are evaluated together; Things Composition & Estimation, Things Discovery, and LSB constitute the IoTS middleware, and are assessed as a whole; XSB and EasyESB are evaluated independently, but also, for part of its evaluation, XSB executes on top of EasyESB.

Hence, we describe, next, AoSBM, Cloud Enactment Engine, IoTS middleware, XSB, and EasyESB evaluations, beginning with use-case-based ones and finally presenting experiment-based evaluations.

4.1. Use-Case-Based Evaluation

The three CHOReOS use cases served as a test-bed to demonstrate the use of the CHOReOS middleware. Table 4.1 shows the use cases in which each middleware component or group of components was validated.

Middleware Component	UC-WP6 Airport	UC-WP7 ACRB	UC-WP8 DynaRoute
AoSBM Service Discovery	×	×	
AoSBM Service Substitution	×		
Cloud Enactment Engine	×	×	×
IoTS middleware	×	×	×
XSB	×		×
EasyESB	×	×	×

Table 4.1: Use of the middleware components by the use cases

4.1.1. IoTS Middleware Use-Case-Based Evaluation

The IoTS Middleware, including the Things Composition & estimation, Things Discovery and LSB, has been widely used by the three CHOReOS use-cases (UC-WP6, UC-WP7, UC-WP8) to facilitate the discovery and access of the Things-based services that take place in the defined scenarios. This qualitative assessment targets to underline the contribution of the IoTS middleware in real-world environments that are exposed through the use-cases.

The UC-WP6 Passenger Friendly Airport use-case concentrates to services provided to air transportation customers. It describes the collection and exploitation of information that appear in an airport environment towards adapting the services offered to travelers and hence, offering a better quality on traveling. The IoTS Middleware is used in this use-case to enable access of sensors appearing both in the environment and the travelers' devices. For example, sound level sensors are used to indicate the noise level at any time in the airport. That supports the automatic adjustment of the sound level during the security announcements at the airport (e.g. louder announcements when the airport is very crowded). Furthermore, location services that are deployed on the passengers' devices are used by the airport authorities to provide directional information, making this way the navigation in the airport spaces more convenient.

The UC-WP7 use-case, named Adaptive Customer Relationship Booster (ACRB), addresses marketing by leveraging the CHOReOS technologies. The IoTS Middleware was integrated to this use-case targeted to provide contextual information of the users (potential customers) to the marketing applications. Specifically, the applications developed in the context of ACRB can use the IoTS Middleware to discover mobile devices of users appearing close to points of interest e.g., a store, and then request information from or take actions on them e.g., send promotional material. Example Things services that are supported are the location and the message delivery service. The first of them is used to expose the current location of the mobile devices and the latter is used by the marketing applications for delivering promotional material to users that might be interested.

The UC-WP8 DynaRoute use-case targets on assisting citizens in the transportation domain. Thing-based services are used in this context to provide contextual information about the stakeholders that take part in the scenario, such as the taxis, the taxi companies and the citizens. Specifically, location services are deployed on the taxi fleet to expose at any time the current location of the taxis to the taxi company and the users. By doing so, the citizens can discover close-by taxis and furthermore, this way the taxi company can manage the fleet more effectively. Additionally, in the same context of use, DynaRoute supports social interactions among users. Location services are deployed on users' mobile devices enabling the DynaRoute application to identify physical proximity between friends.

The flexibility of the IoTS middleware facilitated the discovery and access of the Things services in the aforementioned implementations with minimal effort.Its lightweight nature managed the resource constraints implicated by the mobile devices, which have a significant role in the use-cases, while keeping the performance at a high level. More details on the IoTS Middleware integration in the CHOReOS use-cases can be found in the deliverables related to them, namely, D6.5, D7.5 and D8.5

Despite the fact that the defined use-cases offer a large number of services that could be the required context to perform a quantitative assessment, we concluded that an ultra-large-scale (ULS) evaluation based on simulation, would provide better insights on the performance and scalability of the IoTS Middleware. The procedure taken and the results of this ULS evaluation are presented in Section 4.2.1.

4.1.2. XSB Use-Case-Based Evaluation

The eXtended Service Bus (XSB) has been assessed as part of the CHOReOS runtime environment in supporting the development of the use cases UC-WP6 and UC-WP8. This is a qualitative evaluation and shows the capacity of XSB in enabling the interconnection of heterogeneous services, namely, services that employ middleware platforms which apply different interaction paradigms among the widely used client-server, publish-subscribe and tuple space.

In particular in the UC-WP6 Passenger Friendly Airport use case, XSB is used for including, into the designed choreography of – in their majority – Web services following the client-server paradigm, services following other paradigms. More concretely, an air traffic management notification service using JMS publish-subscribe middleware for signaling a non-anticipated landing of a flight is interconnected with the airport local management Web service. Additionally, an amenities (specifically, in terms of hotel accommodation) reservation service employing a JavaSpaces tuple space middleware for managing accommodation resources is interconnected with the airport local arrangements Web service, which books accommodation for the incoming passengers that have to spend the night.

In the UC-WP8 DynaRoute use case, XSB is used for cross-connecting an airline flight delay notification service, which again employs a JMS publish-subscribe middleware, with the rest of the Web services choreography.

The above implementations were facilitated by the high extensibility of the XSB framework, which enabled developing binding components for all the interconnected services and their heterogeneous middleware platforms with minimal effort. Additionally, putting in place all the necessary adaptations at the business interface and data level of the interconnected services was significantly facilitated by the application development support offered by the XSB framework, which again reduces considerably the developer's effort. Finally, the good performance of the XSB running on top of the EasyESB allowed executing the global choreographies with low latencies that are compatible with the overall system requirements and the user's perception of interactivity whenever the user is involved.

More details about how XSB was used in each of the two use cases can be found in the CHOReOS deliverables related to the final evaluation of the use-cases, namely, D6.5 and D8.5. Moreover, we have carried out a precise quantitative evaluation of XSB in terms of application development support, extensibility, and performance (including testing under stress conditions). The details concerning this evaluation and our results are given in Section 4.2.2.

4.1.3. EasyESB Evaluation

EasyESB is the cornerstone component of the CHOReOS Middleware, consisting in the elementary brick deployed by the Enactment Engine. It is the basis of the EasyESB and XSB distributions and is fully integrated with the Cloud and Grid Middleware. More precisely, EasyESB provides a suitable substrate for the Interaction Binding Components of the XSB middleware. The EasyESB architecture has been refined in order to support the building of Interaction Binding Components atop of it. Furthermore, we have provided interaction paradigms at the levels for Client Service, Publish Subscribe and Tuple Space. Finally, the interaction within Interaction Binding is supported by Generic Application on top of the EasyESB bus protocol.

Consequently, evaluating both components on top of use case and simulation-based conditions involves evaluating EasyESB as well. Thus, we refer to sections4.2.2 and 4.2.6 for a general understanding of the evaluation test bed. Further to these, we have set the following tests based on both use case (WP7) and simulation evaluations. The following Table 4.2 shows the tests, the hardware conditions, and the behavior of EasyESB in such environments. The tests illustrate EasyESB' capabilities at deployment time, and its capacity to scale up in hosting a large number of services on top of each single node. Consequently, connected EasyESB nodes are able to support an ultra large scale number of services without saturating the underlying hardware infrastructure.

Topology	Tests on a single pc including: EasyESB, the services, the GUI and the Profiler.
Initial CPU **Used CPU**	CPU (Intel U7300 @ 1.30GHz) CPU after all the services has been proxified 60% (30% per core)
Initial RAM **Used RAM**	4GB RAM, 3GB available for applications, 1 used by the OS+Desktop 376M when started, 1625M after wrapping all the services
Deployment Parameters	ESB launched with params "-Xmx2048M -Xss128K".
Use Case-based Simulation	60 services for Choreography 7 and 16 services for Choreography 4 (near **80** services).
Simulation-based test	**1100** services deployed on a single EasyESB node.

Table 4.2: EasyESB evaluation

We note also that EasyESB is a standalone bus that does not require the installation of further software such as for IBM and Oracle Solutions. Furthermore, EasyESB nodes can be deployed independently and at different times, forming a dynamic distributed topology. New nodes can join the current topology and nodes can be removed at runtime. By calling the function addNeighborNode(), nodes become aware of each other and can update their list of known services deployed on neighboring nodes.

In Table 4.3 we list the memory usage of several ESBs for the storage and when started with a minimal configuration. The table shows that EasyESB has a small footprint that enables its deployment on top of restricted hardware devices. The modularity of the architecture of EasyESB confers a lightweight footprint that further allows holding only the needed functionalities in a specific distribution. In terms of memory usage, EasyESB can be stored in 50 MB of memory, in comparison with existing solutions, such as Mule ESB with 75 MB, IBM Websphere with 550-600 MB, and 1.1 GB for Oracle Service Bus. When started in its minimal version, EasyESB uses 300 MB, while Mule ESB consumes 768 MB, IBM Websphere uses 1GB to 2GB, and Oracle recommends 4GB.

	EasyESB	Mule ESB	IBM Websphere	Oracle Service Bus
Memory Usage	**50MB**	75MB	550-600MB	1.1GB
Runtime Memory Usage (minimal configuration)	**300MB**	768MB	1to 2 GB	4GB

Table 4.3: EasyESB footprint

4.1.4. AoSBM Discovery and Service Substitution Use-Case-Based Evaluation

The abstraction-oriented service organization and discovery facilities of the AoSBM have been assessed as part of the overall CHOReOS development process to facilitate the development of UC-WP6 and UC-WP7. The way that it has been used is discussed in D2.3 [15]. Briefly, descriptions of services that have been developed in the context of UC-WP6 and UC-WP7 have been registered in the AoSBM. The service descriptions have been organized with respect to abstractions, using the AoSBM clustering facilities. The AoSBM querying facilities have been used in the synthesis process to retrieve information about available services that correspond to the participant roles of the choreographies involved in the use cases. As detailed in [15], the experience from the use of AoSBM in UC-WP6 and UC-WP7 further allowed to assess the AoSBM from a qualitative perspective. In particular, the precision and recall of the AoSBM facilities have been evaluated. Overall, the results showed that the AoSBM is able to mine abstractions that are relevant to the different participant roles, specified in UC-WP6 and UC-WP7, and that the mined abstractions can be used for the discovery of concrete services that are relevant to the different participant roles. The starting point for the assessment of the AoSBM service substitution support was services developed in the context of UC-WP6. We have worked with functional abstraction services, inspired by roles specified in the Passenger Friendly Airport use case. The details concerning this study and our findings are given in Section 4.2.5.

4.1.5. Cloud Enactment Engine Use-Case-Based Evaluation

The Enactment Engine has been used, at least partially, by all three CHOReOS use cases. This usage has substantially helped in improving it, since this actual use aided us in finding bugs, in improving documentation (consolidated on the Enactment Engine User Guide available at the CHOReOS web site), and in prioritizing features to be developed, such as, for instance, the automatic proxification of legacy services through the EasyESB bus, which is an ongoing implementation project to satisfy DynaRoute demands.

The use case with more concrete results up until now in using the Enactment Engine is the Airport use case. The Enactment Engine has enabled the deployment of airport services on the Amazon cloud in an automated way. The Java code produced to enable the enactment of this choreography, by invoking Enactment Engine, encompasses both the choreography specification and the enactment launch. The choreography specification (Appendix, Listing A.1) is assembled with POJOs that follow the Enactment Engine data model, and it has 254 lines describing the choreography, which corresponds to 15 services. If we remove empty lines and method declarations (they exist just to better organize the specification code) we will have in average 11 lines per service. The enactment launch (Appendix, Listing A.2) uses the Java API to invoke the corresponding Enactment Engine operation and is pretty straightforward, counting only around 30 lines.

The very reduced amount of code lines necessary to automate the deployment of each choreography gives an idea of how much work by the deployment team can be saved. Although some work is necessary to properly set up the Enactment Engine, this needs to be done just once; after this step, several choreographies may be deployed taking advantage of the Enactment Engine instance already configured. The Enactment Engine itself may also be configured by third parties and delivered as a service.

4.2. Experiment-Based Evaluation

In addition to the evaluation based on the use cases mentioned above, some key middleware components were also validated based on stress testing and on simulations. These studies are described in the following subsections.

4.2.1. ULS Evaluation of IoTS Middleware

The technical evaluation of the Internet of Things Services (IoTS) middleware proves to be challenging since it is necessary to find an environment which is able to provide the tools for evaluating the *scalability, heterogeneity*, and *mobility support* aspects of the middleware. Specifically, it is imperative to find an evaluation environment that is able to fulfill the following requirements:

- **Ultra-Large scale:** The environment must provide the means for deploying the IoTS middleware in at least 10.000 real or simulated things.
- **Realistic deployment conditions:** The environment must provide the means to deploy the middleware components such as the *Registry Manager* under realistic conditions.
- **Time constraints:** Ultra-Large Scale evaluations must be completed within reasonable time frames, ranging from a few hours up to a few days.
- **Mobility:** The environment must provide mobility, or the means to simulate mobility accurately (e.g. Using mobility traces as input).

In the following, we present the efforts towards the technical assessment of the Internet of Things Services (IoTS) middleware. The remainder of this section is structured as follows. In Section 4.2.1.1 we present a summary of the current assessment methodologies for technical assessment, as well as a short discussion on their applicability to the technical assessment of CHOReOS' IoTS middleware. Section 4.2.1.2 provides a detail description of the chosen assessment strategy and its individual components, before presenting our evaluation results in Section 4.2.1.3, and our conclusions in Section 5.

Background

Traditionally, two main methodologies can be used to perform a thorough technical assessment of a solution: *Testbed-based* evaluation and *Simulation-based* evaluation. In the following we detail the strengths and drawbacks of both methods, before presenting a short discussion on their application for the technical assessment of the CHOReOS IoTS middleware.

Testbed-Based Evaluation Over the years, testbed-based evaluation has proven to be a powerful tool for assessment in different domains. Perhaps its main advantage is that it provides the means for performing tests in *real-time*. In the particular field of IoT, it also enables an easy evaluation of mobility and heterogeneity aspects. On the other hand, when owned, testbeds are expensive to build in both time and money. Moreover, because of these characteristics, scalability is limited.

However, because of the growing number of worldwide research projects oriented towards building the Future Internet, there are a number of publicly accessible testbeds. The largest one deployed so far is SmartSantander [27], which provides a city-scale testbed, currently composed of around 2000 ZigBee sensor nodes. Other large scale deployments include SensLAB [19] with around 1024 sensor nodes, both wired and wireless, across 4 different sites; KanseiGenie [30] with around 700 wired and wireless sensor nodes; and WISEBED [5] with 500 wireless and wired sensor across 5 federated testbeds. Even though some of these projects already provide large-scale numbers, they are still far from the minimum objective of 10,000 nodes.

Other research efforts attempt at lowering the deployment costs by involving both government and industrial partners. Such is the case of PhoneLab at the University of Buffalo [25], where 200 Smartphones where handed to students, staff, and faculty with the support of the NSF and Sprint. A different strategy is adopted by LifeMap at Yonsei University [31], where researchers made available a mobility monitoring application on the Android Market in an attempt to obtain crowdsourced mobility data. Even though this projects propose some interesting ideas for achieving ultra-large scales, they can only do so in the longterm.

Simulation-Based Evaluation Often neglected in the Computer Science field, simulation-based evaluation is a flexible tool that allows faster modeling and validation at lower costs. Moreover, it provides good scalability depending on the simulation tool and the CPU resources of the hardware it is deployed on. However, the main challenge is to find both tools and models that yield sufficiently realistic results.

Throughout the years, the Networking community has developed different tools to provide realistic evaluation through simulation. Perhaps the most well-known is ns-2 [23], which is a discrete event simulator aimed at networking research. Written in OTcl and C++, it is a widely used tool in the networking community with a great number of modules available for it. Another well-known discrete event simulator is OMNeT++ [33], which is entirely C++ based and also provides a large number of modules, with some good wireless sensor network simulators built on top of it. Nevertheless, these two tools are unfit for evaluating Inria's IoTS middleware, since they only allow the deployment of applications written in C++. A less well-known tool is JiST (Java in Simulation Time) [1], developed at Cornell University. It was specifically designed for running unmodified Java network applications, and for providing a scalable simulation environment able to run on commodity hardware. Furthermore, as part of the same project, SWANS, a wireless ad-hoc network simulator was built on top of it.

These simulation frameworks include basic mobility models, such as Random Waypoint [3], for allowing users validate their solutions in dynamic environments. They also allow the use of mobility datasets as input. Unfortunately, publicly available large-scale mobility datasets are scarce, even in well-known dataset sources such as CRAWDAD [29]. Because of the limitations of these simple models, and the limited availability of large-scale mobility traces, some research efforts from the Traffic engineering and vehicular communications communities point towards developing tools for simulating macroscopic and microscopic mobility in both urban and rural areas. Such is the case of SUMO [2] and VanetMobiSim [22], which not only allow the simulation of moving vehicles, but also pedestrians. Moreover, they allow importing actual maps for easily building large-scale mobility scenarios in real urban areas. Furthermore, the output of these simulations may be used as input for the simulators described above.

Discussion As discussed in Page 37, testbed-only evaluations are not inline with CHOReOS' needs, because of scale or time limitations. On the other hand, scalable simulation environments such as JiST, are good candidates for the ULS evaluation of CHOReOS IoTS middleware. In such an environment, the middleware could be evaluated in networks with millions of communicating devices. However, the ULS aspect of the evaluation does not only concern simulating large numbers of smart things, but also the scalability of crucial components such as the *Registry Manager*. Along those lines, question on whether or not the simulator can "fairly" simulate these components appears naturally. Inside a simulation environment, there is no actual way of ensuring resource allocation to simulated entities as it would be in the real world, where the *Registry Manager* would be a powerful machine with dedicated

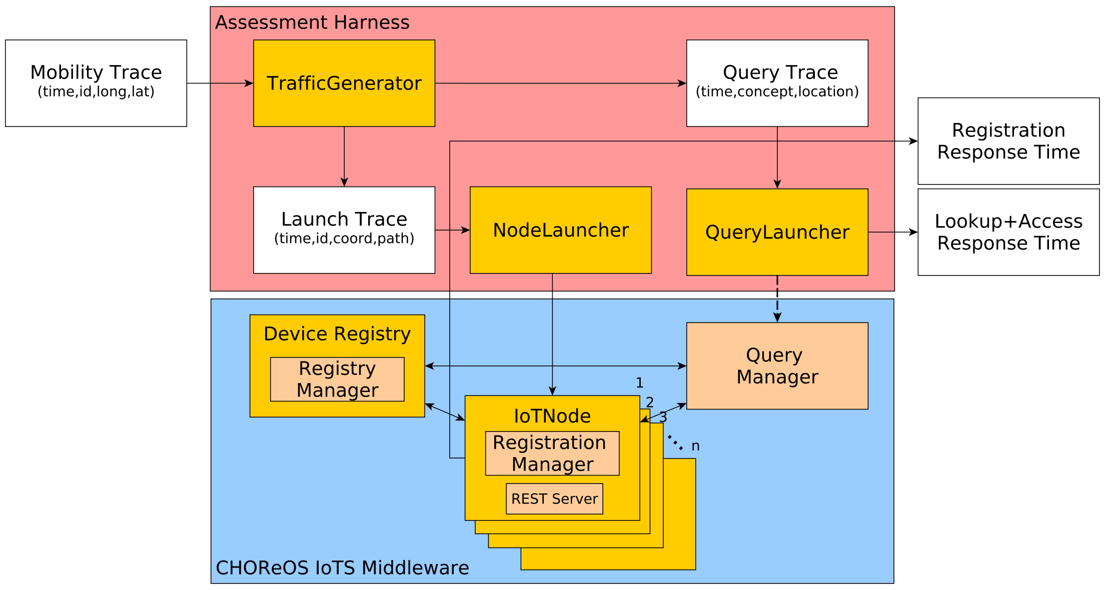


Figure 4.1: Global IoTS evaluation environment architecture

resources. Moreover, a “hybrid” approach where simulating entities residing inside the simulation environment communicate with a real-life entity, is not possible since simulated entities would be running in “simulated time”, whereas real-life entities would be running in “real time”. In other words, time will be “slower” for simulated entities than for real-life entities, thus providing incoherent results. Therefore, simulation-only evaluations are not fit either for the assessment of the IoTS middleware.

Assessment Strategy

To tackle the issues raised by testbed-only and simulation-only evaluation approaches, we propose an assessment strategy in which we combine *offline* mobility simulation and network traffic generation with *online* real-time testbed evaluation. We propose the use of synthetic or real mobility traces to obtain mobility data and generate network traffic. This information is then used to create emulated devices that will interact in real-time with other middleware components deployed on a testbed. In the following, we first present the overall description of such an environment. The details of each component follow thereafter.

Global System Architecture The global architecture of the proposed evaluation environment is depicted in Figure 4.1. There are five main elements composing the system that can be divided into two categories: Assessment Harness Elements, and CHOReOS IoTS middleware elements.

- **Assessment Harness Elements:**
  - **TrafficGenerator:** It’s the main element of the Assessment Harness. It generates network traffic based on data from a mobility trace, and it calculates the mobility of each device, represented by the path (list of times and coordinates) the device will follow over time. It generates a Launch Trace to be used by the NodeLauncher, as well as Query Trace to be used by the QueryLauncher.
  - **NodeLauncher:** Creates, launches, and stops instances of IoTNodes based on the Launch Trace generated by the TrafficGenerator.
  - **QueryLauncher:** Generates lookup and access queries based on the Query Trace generated by the TrafficGenerator. It is able to launch queries concurrently using the *Query Manager* component and measure their response time.

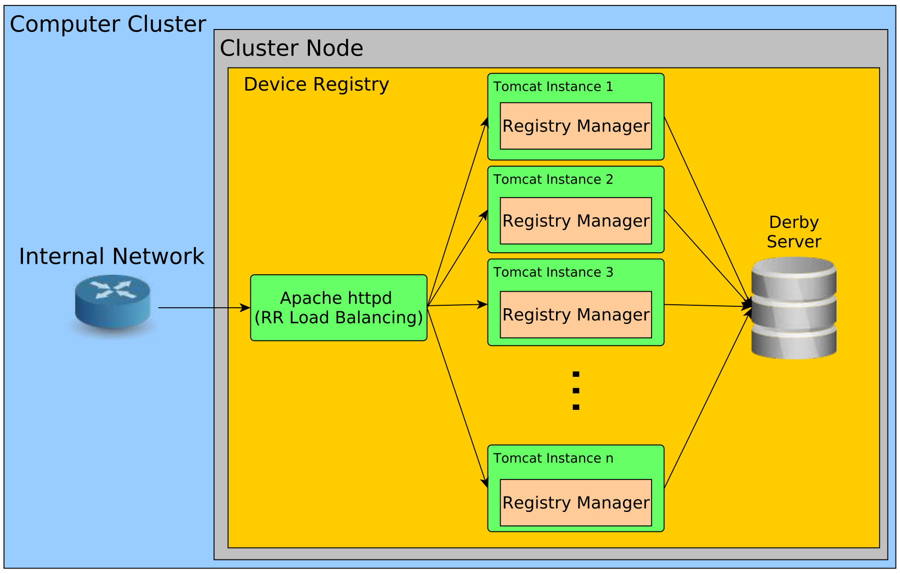


Figure 4.2: Example deployment of the device registry

- **CHOReOS IoTS Middleware Elements:**
  - **Device Registry:** Deploys the *Registry Manager* component under realistic conditions. It stores information on the services provided by the devices, including a network address for each service and relevant metadata. It is also able to provide service metadata on-demand.
  - **IoTNodes:** These elements emulate devices running the *Registration Manager* component. They are created by the NodeLauncher. Each device performs a service registration query and measures its response time. Moreover, each registered device will execute a service corresponding to the one it has registered.

In our assessment environment, each element runs as an independent process. This allows distributing the elements in a computer cluster, thus yielding its large computing resources, and enables our environment to perform ultra-large scale evaluations. The execution of the environment is controlled by different shell scripts that allow launching and stopping the different elements, and fine tuning system parameters, such as the memory used by the processes. In the following, we describe in detail each element composing our assessment environment.

Device Registry: Storing Service Information One of the primary concerns raised by a simulation-only evaluation approach, is its ability to provide a fair evaluation of the *Registry Manager*, a component which, under real life deployment conditions, would be allocated to dedicated computing resources. Therefore, the first step we undertook was to deploy the component as it would be deployed in a production system, thus ensuring adequate resource provisioning.

To closely resemble production systems, the Device Registry is setup as a three-tier architecture such as the one depicted in Figure 4.2. In this setup, incoming requests are received by an HTTP web server (e.g. Apache HTTP) that performs simple load balancing through Round Robin Scheduling, and forwarded to a cluster of application servers (e.g. Apache Tomcat) hosting the *Registry Manager*. Once the registration request is processed, the registration information and metadata are stored in a networked database (e.g. Apache Derby).

TrafficGenerator: Mobility and Network Traffic Generation The TrafficGenerator is the main element of the assessment harness. It obtains mobility information from either synthetic or real mobility

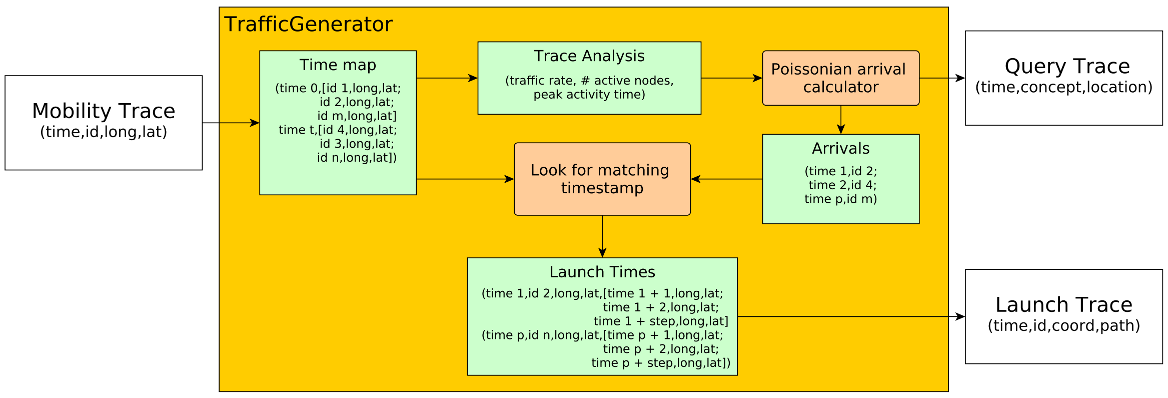


Figure 4.3: Internal operations of the NodeLauncher

traces, and it generates network traffic. In order to be used by our system, input mobility traces are required to have the following format:

Timestamp,Device ID,Longitude,Latitude

Usually, mobility traces only contain the positions of a set of IDs over time. Therefore, no information on network traffic that can be obtained directly from them. Thus, to generate traffic, the TrafficGenerator parses the mobility trace to obtain a time map, where, for each timestamp on the trace, there is a list of device IDs and their coordinates. Based on information from this map, the TrafficGenerator will analyze the trace in order to find the time(s) in the trace at which a user-inputed traffic rate (lambda) is satisfied. Based on this information, the TrafficGenerator calculates a Poisson arrival process to simulate Internet session traffic [4]. At each Poisson arrival, represented by a timestamp, the TrafficGenerator will look in the map for devices with a matching timestamp, and their coordinates at the time. If the coordinates of a node at a particular timestamp are not available, the TrafficGenerator will look for the latest timestamp at which the node's coordinates where known, and calculate the position of the node at the desired timestamp. Then, for each device, a list of subsequent timestamps and coordinates is obtained. by linear interpolation. This list represents the path the device will follow. The length of this list is defined by the user at the start of the evaluation. Finally the arrival and path information is placed in a Launch Trace file that is used by the NodeLauncher to create each individual device, and which has the following format:

Timestamp,Device ID,Longitude,Latitude,Path

The TrafficGenerator also provides the input for the QueryLauncher. Based on the generated Launch Trace, particularly in the number of individual Device IDs in it, and the lowest and highest timestamps, the TrafficGenerator will generate a Query Trace that will serve as an input for the QueryLauncher. As for the Launch Trace, the Query Trace follows a Poisson process, and is written with the following format:

Timestamp,Concept,Location

Both of these procedures are depicted in Figure 4.3. Once the TrafficGenerator provides its output traces, both devices and queries will be launched by the control scripts.

NodeLauncher and IoTNodes: Service Registration Once the mobility and network traffic is generated, devices are created. In our environment, each device is an independent lightweight process that we call an IoTNode. This allows distributing the execution of a large number of devices across multiple machines, thus facilitating the scalability of the simulations. Each IoTNode process is launched by the NodeLauncher based on the Launch Trace generated by the TrafficGenerator.

When the IoTNode is launched, it will attempt to register its service(s) via the instance of the *Registration Manager* component deployed on it. To this end, it will use the information on the type of sensor it possesses, its current coordinates, the path it will follow, and a coverage threshold defined at the start of the simulation. The *Registration Manager* will decide whether the emulated device should register or not based on the coverage threshold, and the macroscopic mobility model defined in the IoTS Middleware (e.g. Truncated Levy Walk). If the registration decision is positive, the *Registration Manager* will register the device for a duration calculated using the device's path, and the IoTNode sets up a REST server to handle requests for accessing its service(s). The address of each service is made up by the address of the host machine, a port number assigned by the NodeLauncher, and the name of the service. Once the REST server is setup, the IoTNode will calculate its lifetime based on its path. At the end of its lifetime the IoTNode will stop its REST server and the lightweight process will stop. Each generated device will generate as output several different CSV formatted files:

- A main file containing its Device ID, the result from the registration query, a timestamp for when the registration request was made, and a timestamp for when the registration response was received.
- A find file containing its Device ID, and the timestamps at which the device contacts the registry to find out how many devices are already registered.
- A calculation file containing its Device ID, and the timestamps at which the registration decision calculations.
- A registration file containing its Device ID, and the timestamps at which the device contacts the register in order to register itself.

At the end of the simulation, each group of files will be merged into a single one according to their type for facilitating their analysis.

QueryLauncher: Service Lookup and Access The final element in our environment is the QueryLauncher. This element will generate sensing queries to obtain information of a particular physical concept on a desired geographical location. As in the case of device generation, sensing queries follow a Poisson arrival process, which is provided as input in the form of a Query Trace.

At each arrival extracted from the Query Trace, the QueryLauncher will create a thread. In each thread, a query is generated with the concept and location information from the trace, for then being passed to the *Query Manager* component. Creating threads allows the QueryLauncher to perform concurrent queries. The *Query Manager* component will then look in the Device Registry for devices hosting the requested service, and access them to obtain a numerical answer to the query. Each Query thread will generate as output several different CSV formatted files:

- A main file containing the requested physical concept, the number of services to access, two timestamps when the query is launched and a response is received, and the result from the query.
- A lookup file containing a thread number, and the timestamps at which the thread contacts the registry to find out the information of the services providing the requested concept.
- An access file containing a thread number, and the timestamps at which the thread attempts to access each IoTNode to retrieve data.

At the end of the simulation, each group of files will be merged into a single one according to their type for facilitating their analysis.

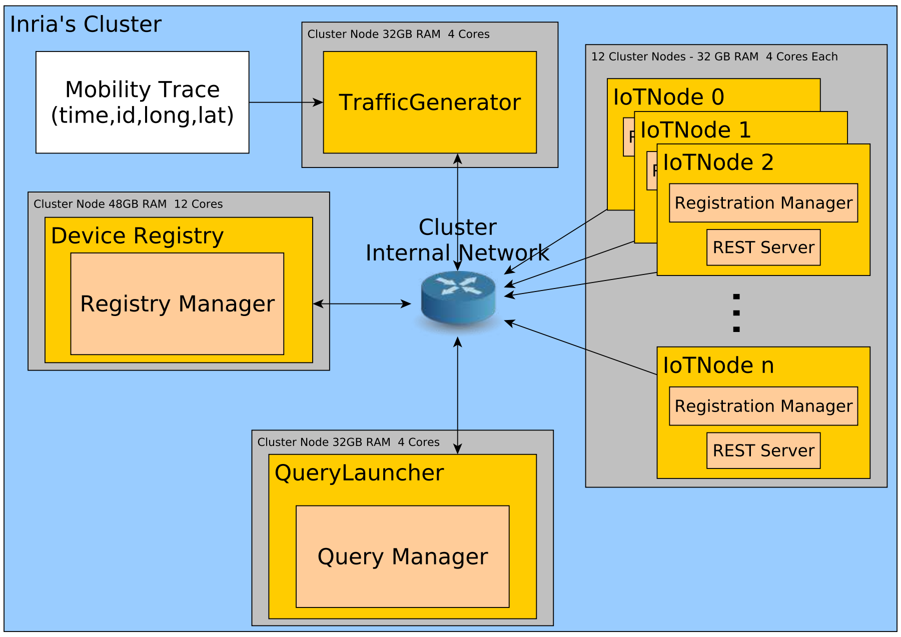


Figure 4.4: Deployment used for performing ULS evaluations

Known Limitations The current version of our evaluation environment presents a number of limitations introduced by the hardware and operating system on top of which the evaluation is performed.

The main limitation introduced by the hardware that supports the evaluation environment is memory availability. The memory available in the machine will limit the number of IoTNode processes that can run concurrently. Similarly, the configuration of the OS on which the evaluations are run may impose limits on the number of concurrent processes a user might run.

A different limitation comes from the maximum number of TCP ports available at an IP address. Even if the machine the evaluations are run on had unlimited memory resources, and the OS was configured to support unlimited processes for a user, each IoTNode needs to bind to a TCP port to provide access to its service. Therefore, the maximum number of available TCP ports imposes a limit on the number of IoTNodes that can be run concurrently on a single machine.

Evaluation Results

This section presents the evaluation results obtained by our evaluation environment. The main objective of our evaluations is assessing the ability of the system of handling large loads of registration and lookup, and access queries. To this end we measure the response times of both types of queries for both *deterministic* and *probabilistic* registration, and *deterministic* and *probabilistic* lookup. We performed these evaluations using a synthetic mobility trace generated with SUMO [2], which uses real mobility data for generating the individual mobility of over 120.000 vehicles during a period of two hours in the city of Cologne [32]. To our knowledge, this is the largest realistic mobility trace freely available at the time of writing. We deployed our system in Inria's computer cluster. The system's elements were distributed inside the cluster to maximize resource utilization. The deployed architecture is depicted in Figure 4.4.

For our experiments we conducted two sets of results. The first set aims at comparing the performances of a system using deterministic registration only, with a system where the CHOReOS IoTS middleware is deployed, and therefore probabilistic methods are used for smartly reducing the number of devices to handle. For this set of results, we varied the input rate to the system from 100 requests per second up to 1000 requests per second. The results show averages from 20 runs and we have calculated the 95% confidence intervals. As input, we used our trace analysis tool to generate traffic

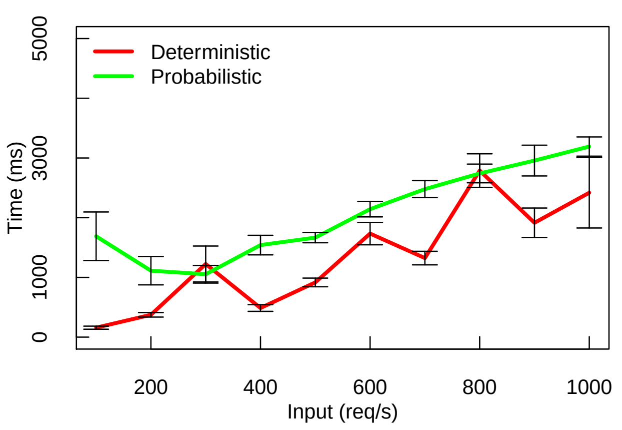


Figure 4.5: Comparison of registration response times for different input rates

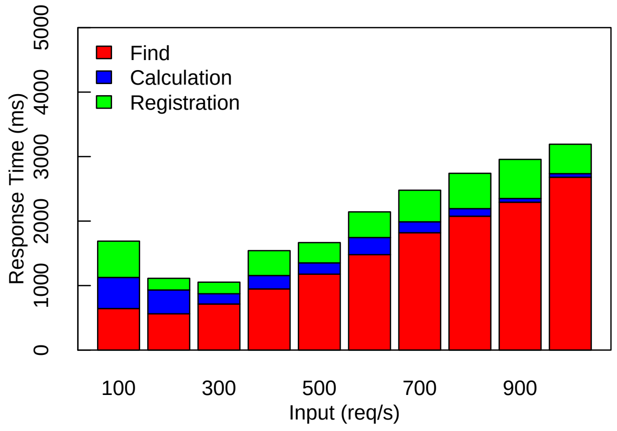


Figure 4.6: Time analysis of probabilistic registration for different input rates

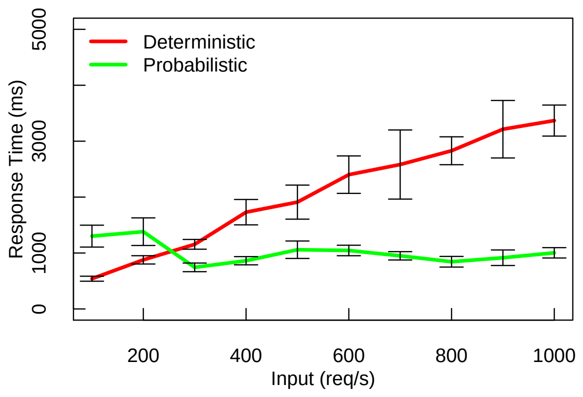


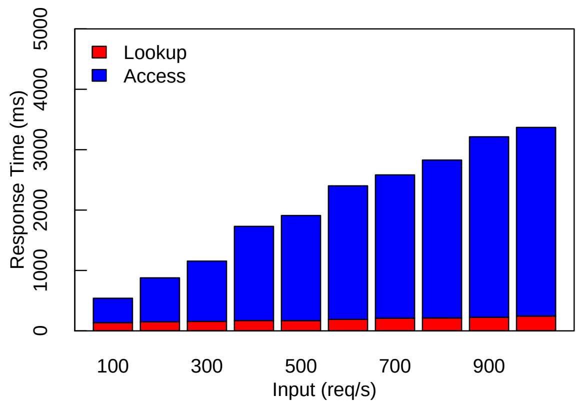
Figure 4.7: Comparison of query response times for different input rates

from most populated sections of the original Cologne trace [32].

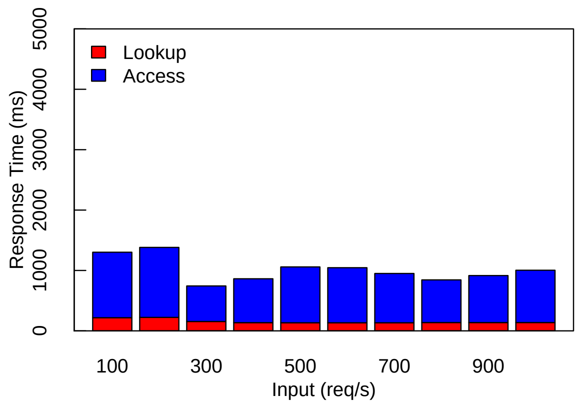
We first measured the response times using both deterministic registration and Probabilistic registration with a coverage threshold of 80%. Figure 4.5 presents these results. Clearly, when the system uses deterministic registration, response times are lower than when using probabilistic registration since no calculations for reaching a registration decision are involved. Nevertheless, probabilistic registration still delivers reasonably fast response times.

In order to better understand the impact of the operations involved in probabilistic registration on the total response time, we calculated the response time of each one of the operations. These are the *Find* operation, which retrieves the number of already devices offering the same service that a device is attempting to register; the *Calculation* operation, which conducts the necessary calculations to generate a registration decision; and the *Registration* operation, which effectively attempts to register a device that has reached a positive registration decision. The results are depicted in Figure 4.6. As seen in the figure, the response time of both *Calculation* and *Registration* operations, is somewhat stable with respect to the input rate, taking around 200 ms in both cases. Nevertheless, the response time of the *Find* operation grows as the input grows. This behavior is due mainly due to two different reasons. First, as the input grows, the number of devices concurrently accessing the Device Registry grows as well, thus having a negative impact on the response times. Second, as the number of already registered devices grows, the time needed for retrieving this information from the Device Registry database is larger. However, this delay is dependent solely on the search algorithms implemented on the database.

We next proceeded to measure the response times using both deterministic Lookup and Access and Probabilistic Lookup and Access with a coverage threshold of 80%. Figure 4.7 clearly shows the impact of the smart probabilistic methods employed by the CHOReOS IoTS middleware on query response



(a) Query response times using Deterministic Lookup and Access



(b) Query response times using Probabilistic Lookup and Access

Figure 4.8: Time analysis of query response times for different input rates

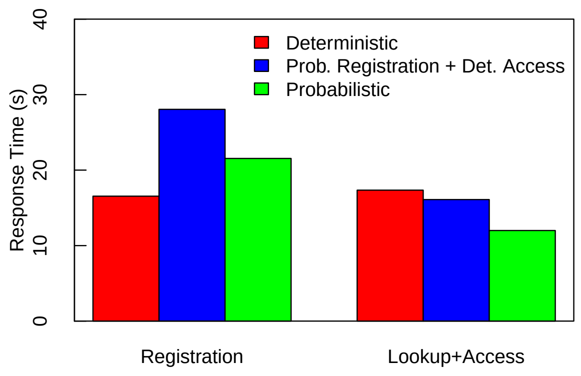


Figure 4.9: Comparison of registration and query response times for ∼15000 nodes at an input rate of 1000 requests per second

times. When using deterministic lookup and access, the system retrieves the full set of registered devices at each query request and attempts to access them all in order to provide a result. Therefore, response times grow larger as the number of registered devices, and the number of query request increases. On the other hand, the benefits of using probabilistic methodologies is clearly shown. In this case, response times are lower and are kept rather stable with respect to number of registered devices and the number of query requests, since the system retrieves only a small subset of registered devices based on the desired coverage threshold.

This is further confirmed by studying the individual behaviors of Lookup and Access for both the deterministic and probabilistic methods, as depicted in Figure 4.8. In both cases, the Lookup response time is very low, and is rather stable for all input rates. In the case of deterministic Lookup and Access, the Access response times grows as the number of registered devices grows, as the system attempts to access the full set of registered devices. In contrast, by using probabilistic Lookup and Access, the Access response times are stable, and are lower than the results of deterministic Lookup and Access specially for high input rates, when a higher number of devices is registered. Note that, the registration strategy has an impact on query response times, since it will indirectly determine the number of devices the query must look for and access.

The second set of results aims at measuring the performance of a the system under ULS traffic for a longer period of time. To this end, our trace analysis tool generated traffic at a rate of 1000 requests per second during 15 seconds based on the information from the Cologne mobility trace [32]. For this set of results, we evaluated a system using only deterministic registration, lookup and access, a system using probabilistic registration and deterministic lookup and access, and a system using the full CHOReOS solution (probabilistic registration, lookup and access).

In Figure 4.9, the response times of both registration and service query operations for the three described systems are shown. In the case of registration, as expected the response times of a full deterministic strategy are lower than the systems using probabilistic registration only. Moreover, the additional overhead introduced by the operations involved in probabilistic registration is clearly shown. In the case of Lookup and Access, as expected, the response times of the systems using deterministic access is higher than the system using probabilistic only. Furthermore, the impact of the registration strategy on service query response times is shown, as the system using probabilistic registration and deterministic access presents a lower response time than the system using deterministic registration only.

CONCLUSION: Overall, these results show the ability of our proposed assessment strategy to deploy the IoTS middleware under realistic conditions, and its capacity to provide meaningful results for the assessment of the solution. Moreover, the obtained results show the ability of the IoTS middleware to smartly tackle the ULS challenge.

4.2.2. XSB Evaluation

We evaluate the eXtended Service Bus (XSB) framework and runtime with respect to two criteria: First, the support that the XSB framework offers to developers – both application developers and middleware developers – when developing a new complex application that is composed as a choreography of a set of heterogeneous services, or when deploying a service, which requires development of an appropriate binding component for the middleware technology that the service is based upon. Second, the performance of the XSB runtime in terms of latency and throughput, under both low traffic and stress conditions. For our evaluations, we rely upon the *Search and Rescue* scenario that we quote below from [21]. This scenario prescribes interconnections between all three interaction paradigms of interest, i.e., client-service (CS), publish-subscribe (PS) and tuple space (TS), within a service choreography that precisely follows CHOReOS principles.

> Search and Rescue (S&R) operations after a disaster, such as a flood or earthquake, are carried out in hazardous environments and require personnel from multiple agencies (e.g., fire-fighters, police) to coordinate. To detect survivors, sensors are installed at various places of the hazardous area. Such sensors communicate their location. S&R personnel also notify at short intervals of their current positions via their PDAs. Upon sensing some life sign, sensor nodes send out notifications. At the same time, nearby light-emitting actuators start lighting the place to facilitate the rescuing effort. Sensors, PDAs, and actuators interact among them and with external actors via a tuple space. Tuple space location and life sign data are sent via client-service invocations to a planning service that recommends at real time the optimal deployment of rescue forces. This output is notified via a publish-subscribe system to the coordinator of the operation on her smartphone and also to a number of control/monitoring centers. The coordinator may approve and command S&R personnel via the publish-subscribe system and the tuple space system to rush into the spot.

We have implemented the S&R scenario on top of the XSB framework. Our scenario implementation integrates: (1) sensors, actuators and personnel equipment communicating over a Jini JavaSpaces TS[1]; (2) the planning service implemented as a JMEDS DPWS Web Service[2]; and (3) a JMS PS system based on Apache ActiveMQ[3] that the coordinator of the operation uses to receive recommendations

[1] http://www.jini.org/wiki/JavaSpaces_Specification
[2] http://ws4d.e-technik.uni-rostock.de/jmeds/
[3] http://activemq.apache.org/

and to send commands. We provide support for the three mentioned middleware platforms by producing appropriate binding components. Based on this implementation, we evaluate the XSB with respect to the two criteria introduced above. We present our evaluation results in the following.

Support to Developers

The XSB framework can be used by both application and middleware developers. We evaluate first the effort for the application developer and accordingly the provided support by our solution for developing complex applications from the integration of services that employ heterogeneous interaction paradigms. Second, given that we have designed our architectural framework with particular consideration for its extensibility, we evaluate the easiness for the middleware developer in integrating new middleware platforms, in particular with regard to building related binding components (BCs).

Application Developer – Effort for Application Design Table 4.4 summarizes our measurements of the development effort required for the S&R scenario. Essentially, this effort includes writing an xDL description for each constituent service, and providing mapping directives between the data exchanged among the services. GA-IDL service descriptions are then generated automatically by using the tools provided by our platform. We see that application development effort is considerably low, since our platform takes care of resolving the interaction paradigm and middleware heterogeneity among the constituent services.

	xDL description (XML lines)	Generated desc. (XML lines)	Mapping directives (XML lines)
Java Spaces system	148	98	72
DPWS system	50	61	76
JMS system	209	90	78
Total	407	249	226

Table 4.4: Development effort for the application developer

Middleware Developer – Extensibility Referring to the architectural framework of Figure 2.3, we measure the effort for building a BC for the JMS Apache ActiveMQ middleware platform. Table 4.5 summarizes this effort, in terms of implemented numbers of: (1) Lines of code, (2) XML schema lines regarding the xDL descriptions, and (3) XML lines of configuration files for the architectural framework. We have performed our measurements with the Metrics 1.3.6 Eclipse plugin[4]. We provide measurements for each one of the three components of the framework, as well as the ratio of the effort specific to the JMS platform (refinement of subcomponents) over the total effort (i.e., including the generic code written once and reusable each time). We see that considerably small effort, no more than 6% of the total effort, is required for the integration of a new middleware platform. This points out the significant support offered, resulting in considerable easiness for integrating new middleware platforms and related high extensibility of our approach.

Performance

The XSB runtime introduces a number of extensions to the typical ESB infrastructure, such as transfer of GA primitives as payload of substrate bus (EasyESB) communication primitives, and, more importantly, runtime model transformations inside the BC; these enable cross-connection and adaptation among heterogeneous interaction paradigms. Hence, we need to evaluate the performance of our solution and the time overhead introduced.

[4]http://metrics.sourceforge.net

	Lines of code	XML schema (lines)	Configuration (XML lines)
xDL Processor	7520	2617	111
Core Engine	9993	219	137
Envelope for Substrate Bus	508	0	0
Total	18021	2836	248
Written by the developer	1162	191	12
Effort	6%	6%	4%

Table 4.5: Development effort for the JMS binding component

We evaluate the performance of the XSB runtime with two experimental setups. In the first one, we employ the XSB with a mock substrate bus, which enables us to assess the pure XSB functionality and to test the XSB under stress conditions without having to internally tune at the same time the EasyESB configuration. We interconnect services employing the same or different interaction paradigms, and measure end-to-end latency and throughput, under both low traffic and stress conditions. In the second experiment, we employ the XSB on top of the EasyESB and evaluate the latency overhead introduced by the EasyESB and by the XSB for various combinations of interconnected interaction paradigms, under low traffic conditions.

End-to-End Performance under Stress Testing We evaluate the performance of the XSB Binding Components under stress conditions relying on [28] and [20]. Our approach enables setting a lightweight testing environment around the XSB with practical hardware resources and making sure that the XSB employs its maximum capacity. In particular, to evaluate the performance capacity of an XSB, the ESB has to be under high load, i.e., we have to saturate the ESB system to determine its maximum performance. To this end, service requestors and service providers need to send and receive high request rate through the ESB.

Test Scenario: We set up our XSB Binding Components with a mock substrate bus and mock services (requestors, providers). In order to test the connectivity of heterogeneous services under stress conditions, we utilize the imported middleware services of the XSB Framework (DPWS, JMS and JavaSpaces services). We make sure to remove any potential bottlenecks from the services (clients/providers) and we create a bottleneck at the service bus; our purpose is to measure throughput and one-way end-to-end latency. More specifically, we use middleware clients (DPWS or JMS) and threads to create many mock service clients. Then, the mock substrate bus that we configure for the evaluation can handle their service requests as messages. We also need to have a service provider that can receive thousands of messages per second in order to overload the XSB. The CPU usage of the machine hosting the XSB should be close to 100% to reach the maximum performance of the XSB. In addition, it is important that both the service clients and providers are not highly loaded.

Test Setup: We used the following software and hardware for our experiments. The setup consisted of four machines, connected via Inria's 100 Mb/s Ethernet network. The first and second machine (M1, M2) have each an Intel core i5-2540M x 2.6 GHz (6 GB RAM), the third (M3) has an Intel Xeon W3550e 3.08 GHz x 4 (7.8 RAM), and the last machine (M4) has an Intel core T7200 2.00 GHz x 2 (2.0 GB RAM). Running tests on powerful machines allows simulating a large number of clients more accurately, as opposed to simple core machines.

We provide four test scenarios:

1) *C - S: Using the DPWS middleware, we create service mock clients and a mock web service running on M1 and M3, correspondingly.*

2) *P - b - Sb: Using the JMS middleware, we create mock publishers, a broker, and a mock subscriber running on M1, M2 and M3, correspondingly.*

3) *C - XSB - S: Using the DPWS middleware and the XSB over the mock substrate bus, we create service mock clients, an XSB service bus, and a Web service running on M1, M2 and M3, correspondingly.*

4) *P - b - XSB - S: Using the DPWS and JMS middleware, as well as the XSB over the mock substrate bus, we create mock publishers, a broker, an XSB service bus, and a web service running on M1, M4, M2 and M3, correspondingly.*

The first scenario doesn't have any service bus, service clients are simply forwarding messages to the web service. The second scenario uses the XSB, and messages pass through it performing a transformation from CS to GA and back to CS primitives. The third scenario doesn't have any service bus, publishers are simply forwarding messages to the broker, and the broker to the subscriber. Finally, the forth scenario uses the XSB and messages are passed through it performing a transformation from CS to GA and from GA to PS primitives (Figure 4.10). Note that, in all four cases, clients make requests every second. In our measurements, we discard the first 4000 messages allowing machines to reach a steady state. Services generally don't exchange very large quantities of data, so messages usually tend to be of average size. Hence, we set the size of messages to 1 KB.

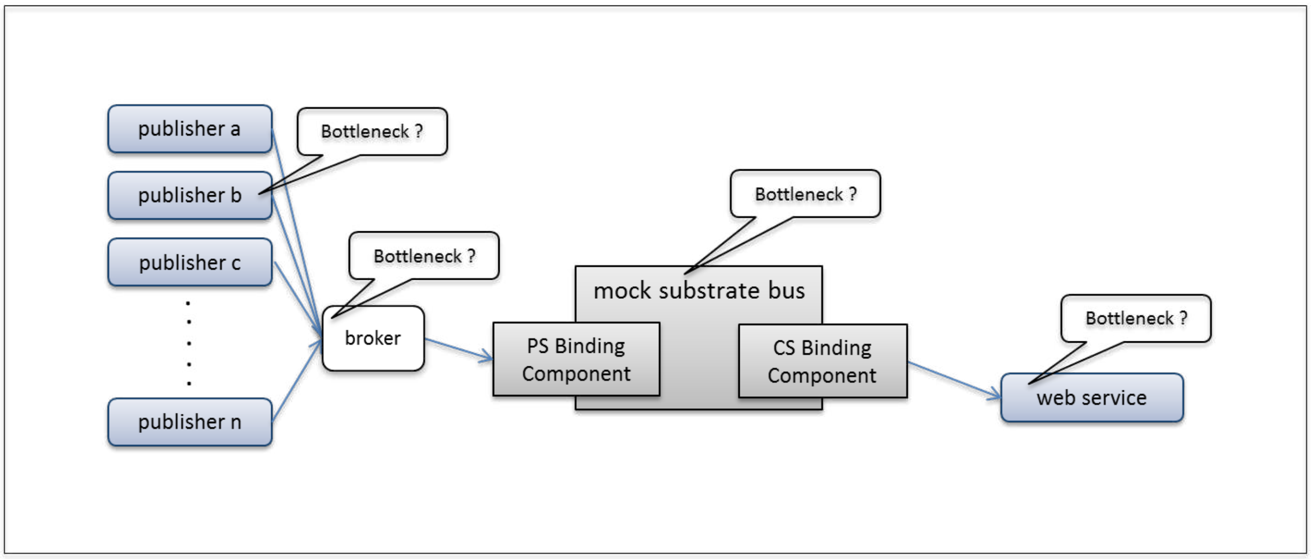


Figure 4.10: Components of mock environment for XSB capacity testing

Results: Table 4.6 presents some detailed data for a test run with 50 concurrent clients. Showing data for each performed test run in this table would be impractical, as it would require several pages.

Scenario	**Latency** (ms.)	**Throughput** (req./s.)	**Avg. Clients**
C-S	2,13	62,78	50
P-b-Sb	1,16	49,92	50
C-XSB-S	17,85	53,9	50
P-b-XSB-S	16,52	56,04	50

Table 4.6: Results for one-way interaction in all four scenarios with 50 concurrent clients

Figure 4.11 shows the measured throughput when sending one-way messages to the services, in function of the number of concurrent clients for each of the above scenarios. The procedure we applied to execute this experiment is the following: i) In all four cases, after the XSB reaches the steady state, service providers are counting incoming messages for a duration of time, and ii) we repeat the same experiment by increasing the service clients. Thus, JMS and DPWS service providers receive an increasing number of messages according to the concurrent clients. We observed that the number of messages passing via the XSB per second (throughput) remains unchanged after 90 concurrent clients. We verified at the same time that the CPU usage of the machine hosting the XSB had reached its maximum, while the CPU for clients/providers is about 15% - 20%. In scenario 1 using the DPWS middleware, the maximum throughput is 220,89 requests per second, while in scenario 2 using the JMS middleware, the maximum throughput is 243,47 requests per second. For communication through the XSB, in scenario 3 the maximum throughput is 82,29 requests per second, while in scenario 4 it is 90.8 requests/sec.

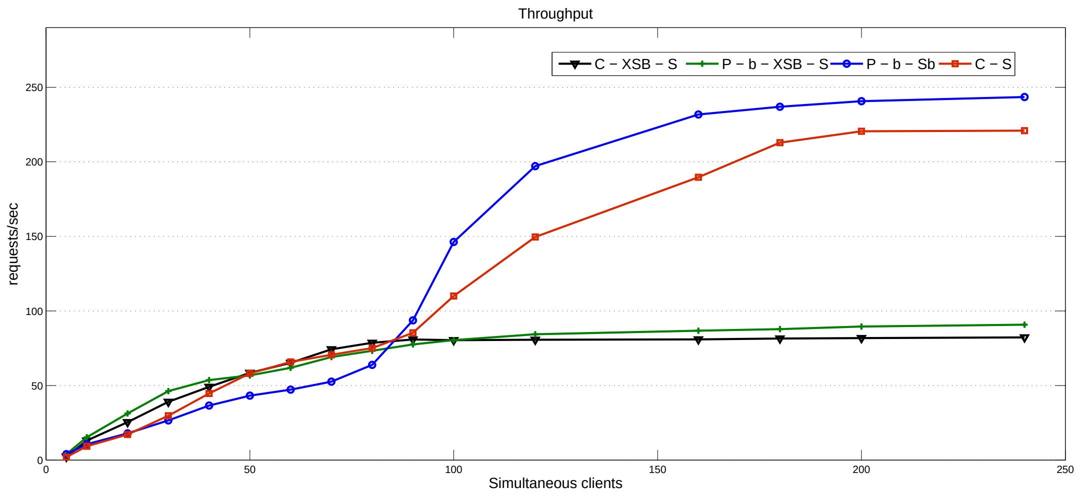


Figure 4.11: Throughput for one-way interaction of DPWS and JMS services with or without XSB in scenarios 1, 2, 3 and 4

Figures 4.12 and 4.13 show the measured one-way latency when making calls on the services, in function of the number of concurrent clients for each of the above scenarios. The procedure we applied to execute this experiment is the following: i) In all four cases, after the XSB reaches the steady state, service clients are sending requests continuously for a duration of time, ii) service providers are active to receive all messages, and iii) we repeat the same experiment by increasing the service clients. We observed that messages passing via the XSB have high latency after 50 concurrent clients. When using the DPWS middleware, the latency is 3 milliseconds, while, when using the JMS middleware, it is 1,8 milliseconds, both for 240 simultaneous clients and without passing messages through the XSB, as shown in Figure 4.12. On the other hand, for communication through the XSB, the latency reaches quite high values (Figure 4.13).

In Figure 4.13, latency values for a low numbers of clients are not clear due to the scale used, thus we depict in Figure 4.14 the measured latency for up to 50 clients. The latency for 50 clients is 17,85 milliseconds when using the DPWS middleware and 16,2 milliseconds when using the JMS middleware.

Considering the fact that the CPU usage of the XSB reached its maximum in both experiments, results show that after 90 concurrent clients, the XSB cannot manage the reached number of clients. In our future work, we intend to optimize the XSB for improving its throughput and latency values in the case of many concurrent clients.

Latency Overhead on the Bus We measure execution times for a number of layouts: (i) one-way and two-way interaction inside our implemented CS system; (ii) end-to-end interaction between a publisher and a subscriber inside our implemented PS system; (iii) end-to-end interaction between a writer and a reader inside our implemented TS system; (iv) one-way and two-way interaction between two CS peers via EasyESB; and (v) interaction between all pair combinations of CS, PS and TS peers via XSB. We repeat each measurement a 100 times and calculate mean values. Based on these experiments, we

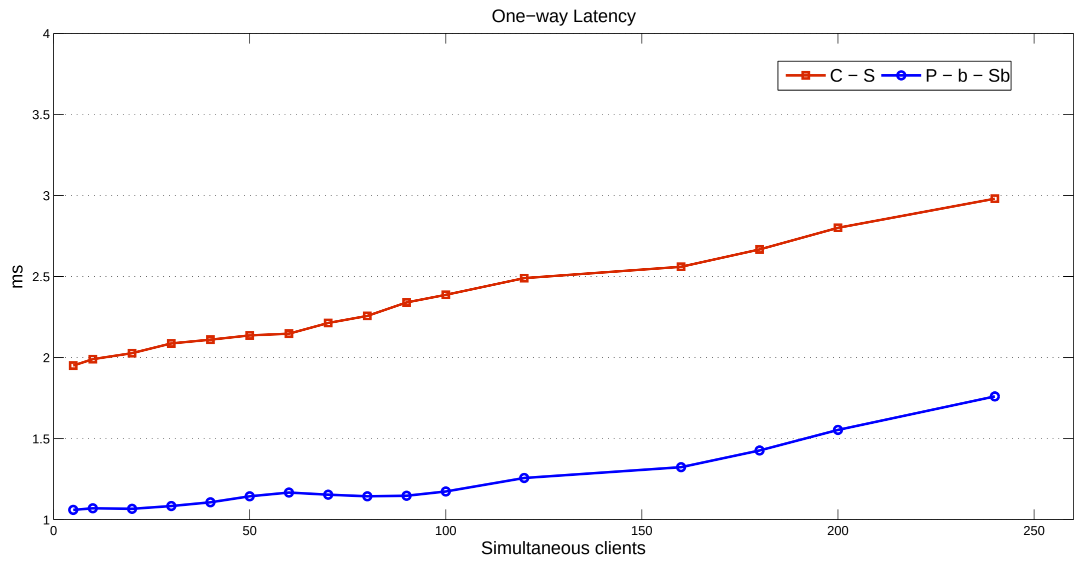


Figure 4.12: Latency for one-way interaction of DPWS and JMS services without XSB in scenarios 1 and 2

evaluate the latency overhead introduced by the EasyESB for an one-way CS-CS communication, and the latency overhead introduced by the XSB for an one-way CS-CS as well as all other pair combinations of communication. Our results are summarized in Table 4.7. We see that the latency overhead introduced by the XSB for a CS-CS interconnection is only 1% greater than the latency overhead introduced by the EasyESB itself. When conversion between heterogeneous interaction paradigms is involved, the XSB latency overhead ranges from 7% to 15,5%, where we note that we always compare with the EasyESB CS-CS homogeneous interconnection, since EasyESB support for other interaction paradigms is not available. We see that the performance cost introduced by the XSB remains at reasonable levels.

Interconnection	Latency (ms)
one-way CS - CS via EasyESB	258
one-way CS - CS via XSB	261,5
CS - PS via XSB	283
CS - TS via XSB	276
PS - TS via XSB	298

Table 4.7: Interaction latency on the bus for each interconnection

CONCLUSION: Our evaluation of the XSB framework and runtime showed quite good results in terms of both developer support and performance. We note that our software engineering support evaluation is based only on counting lines of code for a single developer. A more comprehensive empirical evaluation would require as well a subjective evaluation of the framework facilities by a number of developers. We also point out that stress testing requires very well-thought experimental setups when such a big number of factors intervene, both in hardware and in software in which there are many indirection layers and dependencies on external components.

4.2.3. EasyESB Evaluation

Please refer to section 4.1.3.

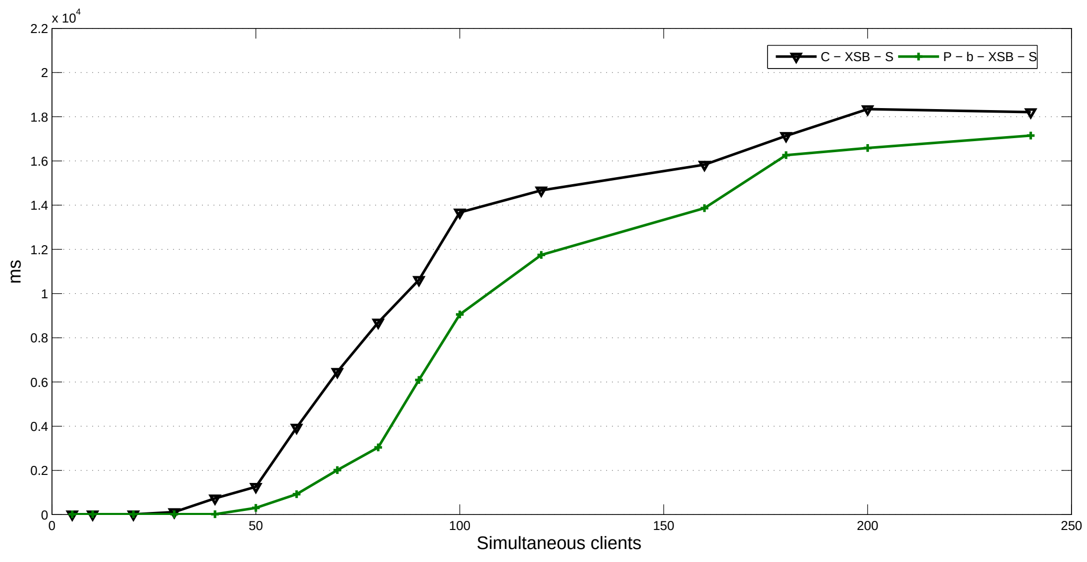


Figure 4.13: Latency for one-way interaction of DPWS and JMS services with XSB in scenarios 3 and 4

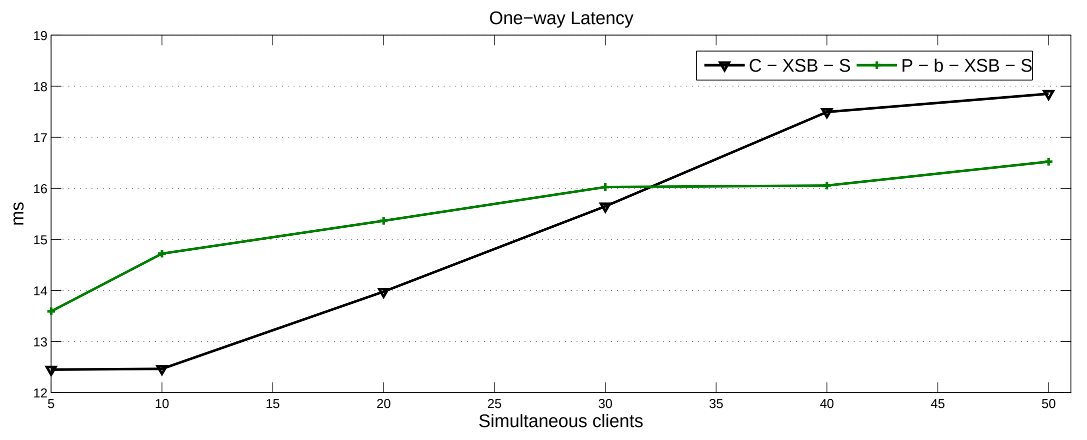


Figure 4.14: Latency for one-way interaction of DPWS and JMS services with XSB in scenarios 3 and 4 for up to 50 concurrent clients

4.2.4. AoSBM Discovery Evaluation

In the CHOReOS deliverable D2.3, we provided a qualitative assessment of the usefulness of AoSBM in the CHOReOS use cases. Here, we provide a quantitative assessment that focuses on the performance of AoSBM in a ULS setting.

Overall, AoSBM is driven by the hypothesis that service abstractions allow for efficient execution of service lookup queries in the context of the Future Internet, where the amount of available service descriptions is expected to dramatically grow. Therefore, the AoSBM evaluation focuses on the following research question:

- Is querying based on abstractions faster than conventional querying, based on concrete service descriptions?

Experimental Setup

To address this question, we compared the time it takes to execute WSBQL queries for weather forecast services (Table 4.8) over service abstractions, stored in the AoSBM relational store, with the time it takes to execute corresponding SQL queries, over concrete service descriptions, which were also stored in the AoSBM relational store.

Table 4.8: WSBQL Query

```
let        $aosbm = db(``localhost/mySB'')
for        $c in $aosbm/servicecollections
for        $sa in $c/hierarchies/abstractions
for        $if in $sa/representativeinterfaces
for        $o1 in $if/representativeoperations
for        $o2 in $if/representativeoperations
for        $m1 in $o1/representativemessages
for        $m2 in $o2/representativemessages
for        $p1 in $m1/representativemessagetypes
for        $p2 in $m1/representativemessagetypes
for        $p3 in $m2/representativemessagetypes
where
           $if/rsi_name like %Weather% and
           $op1/rop_name like %Temperature% and
           $p1/rmt_name like %Location% and
           $p1/rmt_type = `String' and
           $p2/rmt_name like %Date% and
           $p2/rmt_type = `String' and
           $p3/rmt_name like %UnitSystem%
return
           Abstractions.fullObject
```

Our WSBQL queries consist of three main parts:

- The first part consists of variable definitions. The variables define constraints that should be satisfied by retrieved abstractions. In particular, we have the following variables: `$sa` stands for a service abstraction; `$if` corresponds to the abstract interface of `$sa`; `$o1` and `$o2` represent two operations that should be provided by the required abstract interface; finally, `$p1` and `$p2` are two parameters of `$o1`, while `$p3` is a parameter of `$o2`.
- The constraints that should be met by the retrieved service abstractions are specified in the `where` part of the query. Specifically, for operation `$o1`, we have the following constraints: the name of `$o1` should include the term `Temperature`; the name of parameter `$p1` should include the term `Location` and the type of `$p1` should be `String`; the name of parameter `$p2` should include the term `Date` and the type of `$p2` should also be `String`. For the operation `$o2`, there is only one constraint that concerns the name of parameter `$p3`, which should include the term `UnitSystem`.
- The `return` part of the query dictates the information that should be included in the result: with the `fullObject` option, the result contains full information about the retrieved service abstractions and the represented services.

The queries were executed over various AoSBM instances, populated with different synthetic data. More specifically, we organized our experiments in two sets (Table 4.9). In the first set of experiments, we varied the number of abstractions from $5 * 10^3$ to 10^6 and the number of service descriptions from $5 * 10^4$ to 10^7; hence, each abstraction represented 10 services. In the second set of experiments, the

Table 4.9: Experimental Setup

data set properties	AoSBM instances					
# service abstractions	$5*10^3$	10^4	$5*10^4$	10^5	$5*10^5$	10^6
# concrete services	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7
# represented services per abstraction	10					
# operations per service	3					
# in parameters per operation	2					
# out parameters per operation	2					
overall disk space (MB)	159	325	1700	3451	17920	37478

(a) 1st set of experiments

data set properties	AoSBM instances					
# service abstractions	10^4					
# concrete services	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$	10^7
# represented services per abstraction	5	10	50	100	500	1000
# operations per service	3					
# in parameters per operation	2					
# out parameters per operation	2					
overall disk space (MB)	171	325	1587	3205	16486	33382

(a) 2nd set of experiments

number of stored abstractions was 10^4, while the number of service descriptions ranged from $5*10^4$ to 10^7; thus, the number of represented services per abstraction varied from 5 to 1000. Further details concerning the experiment setup for each set of experiments (operations per service, in/out parameters per operation, required disk space) are given in Table 4.9. We executed our experiments on a typical Intel Core 2, 2.2GHz, 3GB RAM; for the AoSBM relational store we employed MySQL server 5.5. We performed each experiment 10 times.

Results & Findings

Figures 4.15 and 4.16 gives the results that we obtained; the reported numbers are the average values.

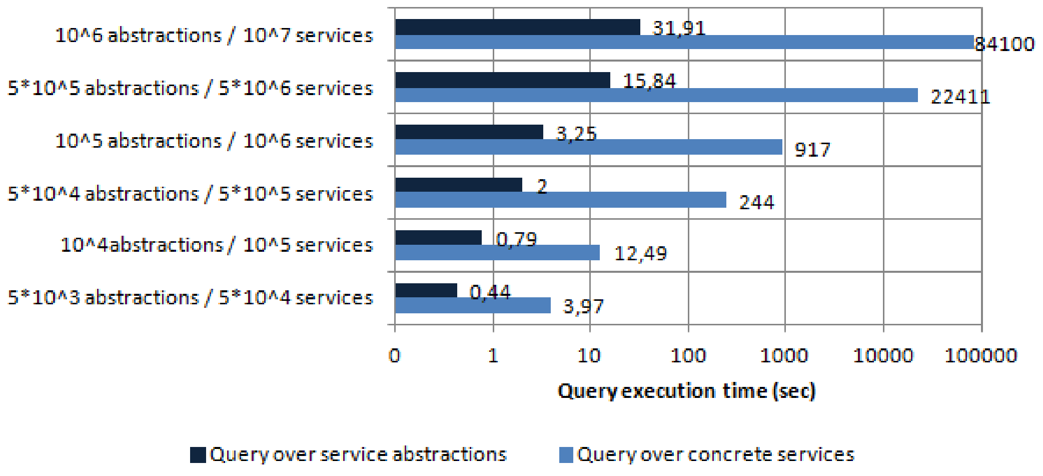


Figure 4.15: Querying service abstractions vs. concrete service descriptions – 1st set

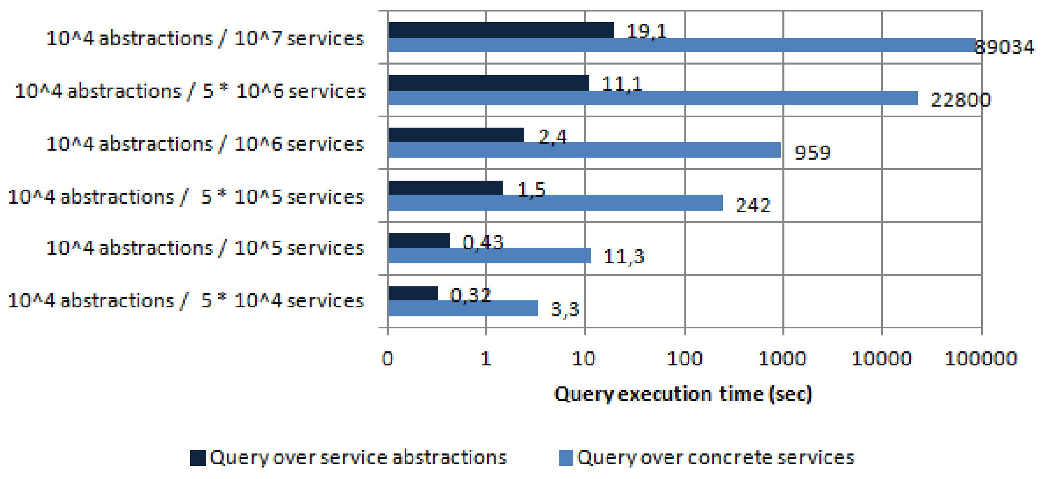


Figure 4.16: Querying service abstractions vs. concrete service descriptions – 2nd set

Our findings are summarized in the following points:

- In both sets of experiments (Figure 4.15 and 4.16), we observe that the SQL query execution time increases with the number of concrete service descriptions stored in the AoSBM relational store.
- Concerning the WSBQL query, in the first set of experiments, the execution time increases with the number of stored service abstractions. On the other hand, in the second set of experiments, the WSBQL query execution time increases with the size of the result; the number of represented services for the service abstractions that are returned by the query varies from 5 to 1000.
- In the first set of experiments, querying over service abstractions is 88% to 99% faster than querying over concrete service descriptions. Similarly, in the second set of experiments querying over service abstractions is 90% to 99% faster than querying over concrete service descriptions.

CONCLUSION: To summarize, the main take-away message from the AoSBM evaluation is that *querying over service abstractions is much faster than querying over concrete service descriptions*; while conventional querying increases the number of available service descriptions, abstractions-oriented querying increases with the number of available abstractions.

4.2.5. AoSBM Service Substitution Evaluation

We have experimentally assessed our AoSBM service substitution approach with different web services and configurations of their invocation. Our evaluation concerns two main aspects. The first aspect is the time overhead that is introduced by the AoSBM service substitution approach for the translation of invocations made on a functional abstraction service, to invocations on the concrete services that are hidden behind the functional abstraction service (Section 2.1.2) [11]. The second aspect is the time overhead that is introduced by the approach for the actual substitution of one concrete service with another (Section 2.1.2) [11]. Regarding these two aspects, we focus on the following research questions:

RQ1: What is the time breakdown for the translation of an abstract invocation to a concrete invocation, with respect to the three different phases of the functional abstraction service delegation mechanism? What is the effect of the input/output size to the translation time?

RQ2: What is the time breakdown for service substitution? How much time is spent to notify the services that use the substituted service about the beginning/end of the substitution and how much time is required to perform the change of the mapping that is used by the functional abstraction service?

In the rest of this subsection we shed some light to the above questions. Before proceeding, however, we detail our experimental setup.

Experimental Setup

RQ1 - Assessing the Delegation Overhead To evaluate the delegation overhead introduced by our approach, we have worked with 6 functional abstraction services, inspired by roles performed within the WP6 Passenger Friendly Airport use case:

- `AbstractWeatherForecastService`: serves for the substitution of weather forecasting service instances.
- `AbstractHotel`: deals with the substitution of hotel service instances.
- `AbstractSecurityCompany`: allows the substitution of security company service instances.
- `AbstractAirplane`: serves for the substitution of airplane service instances.
- `AbstractDisplaysManagement`: enables the substitution of airport notification service instances.
- `AbstractAirportBusCompany`: deals with the substitution of services for the transport of passengers within the airport.

The assessment of the delegation overhead is performed by invoking each functional abstraction service both as a Web service and as a simple Java class. We investigate the delegation overhead and its breakdown in the three different parts of our method for the transformation of abstract to concrete invocations. In the case of the `AbstractWeatherForecastService`, we vary the size of the input (and accordingly, the output) to see its effect over the different parts of the execution. In particular, the input (respectively, the output) parameter is a list and we consider invocations for which the size of the list is 100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000 elements. The rest of the functional abstraction services do not offer us the luxury of modifying their input size (and accordingly, their output). Therefore, in these cases we simply measure the delegation overhead, without assessing the input/output size effect on this overhead. For each measurement, we execute it 10 times and take the average execution time. The deployment for the experimental assessment is as follows: the concrete service instances as well as the functional abstraction services are deployed on a server workstation and the client that invokes the services is located in another PC.

RQ2 - Assessing the Substitution Overhead For the assessment of the substitution overhead, without loss of generality, we focus on the case of the `AbstractWeatherForecastService`. To study the time breakdown for substituting the concrete service instance that is hidden behind `AbstractWeatherForecastService`, with another one we consider different configurations, where the number of services that use the hidden substituted service instance, via `AbstractWeatherForecastService`, varies from 100 to 1000. Therefore, the number of services that are notified, also varies from 100 to 1000.

Results & Findings

RQ1 - Assessing the Delegation Overhead Figure 4.17, gives the delegation overhead breakdown for the `AbstractWeatherForecastService`, when invoked as a simple Java class, while Figure 4.18 gives the delegation overhead breakdown for the `AbstractWeatherForecastService`, when invoked as a Web service. Figure 4.19, gives the overall invocation time in the aforementioned two cases. The results are relatively similar and we can summarize our findings as follows.

- As the size of the input increases (respectively, the output), the total time for the completion of a invocation increases, too.
- When the functional abstraction service is invoked as a Web Service, there is an extra cost incurred compared to its invocation as a Java API, due to the extra dereference level that the Web service incurs.
- The execution time of the two first phases of the translation of abstract to concrete invocations, that require a conversion to the unified intermediate representation and a customization towards the particularities of the actually invoked concrete service, are affected by the size of the input/output data that are translated. The larger the lists we have to manage, the more time we spend for their management and manipulation.
- The second phase of the data translation, both concerning the input and the output takes the lion's share in the delegation overhead breakdown, been found several orders of magnitude higher than any phase of the translation process. This is due to the fact that (a) our mechanism traverses the input or output object using the Java reflection mechanism and (b) it searches in the mappings that have been created with the use of lists, iterating over all the objects in the list until the appropriate object is found, suffering thus from the necessary time penalty that the iteration over large lists incurs.
- The time needed to initialize the input or output objects increases as the input or output object increases.

Figure 4.20 gives the delegation overhead breakdown for `AbstractHotel`, `AbstractSecurityCompany`, `AbstractAirplane`,`AbstractDisplaysManagement`, and `AbstractAirportBusCompany`, when invoked as a simple Java classes, while Figure 4.21 gives the delegation overhead breakdown, when invoked as a Web services. Figure 4.22 gives the overall invocation time in the aforementioned two cases.

The results are relatively similar and we can summarize our findings as follows.

- Once again, we pay a small extra price for the invocation of a functional abstraction service as a Web service, as compared to its invocation via a Java API.
- The second phase of the input data translation is typically (but not always) slower than the rest of the data translation steps. The output restructuring behaves similarly. However, we do not observe differences that are orders of magnitude higher, although there are three out of six times where some restructuring is three times higher than the rest of the parts. Clearly, the size and simplicity of the input and output objects are the responsible reasons for this behavior.

RQ2 - Assessing the Substitution Overhead Figure 4.23, gives the results of the experimental assessment of the substitution overhead. Our observations are summarized below:

- The number of the services that use the substituted service, affects the time needed for their notification from the impact analyzer that the substitution starts. Naturally, this is due to the fact that an increased number of such services requires more invocations, for the commencement of the substitution process.

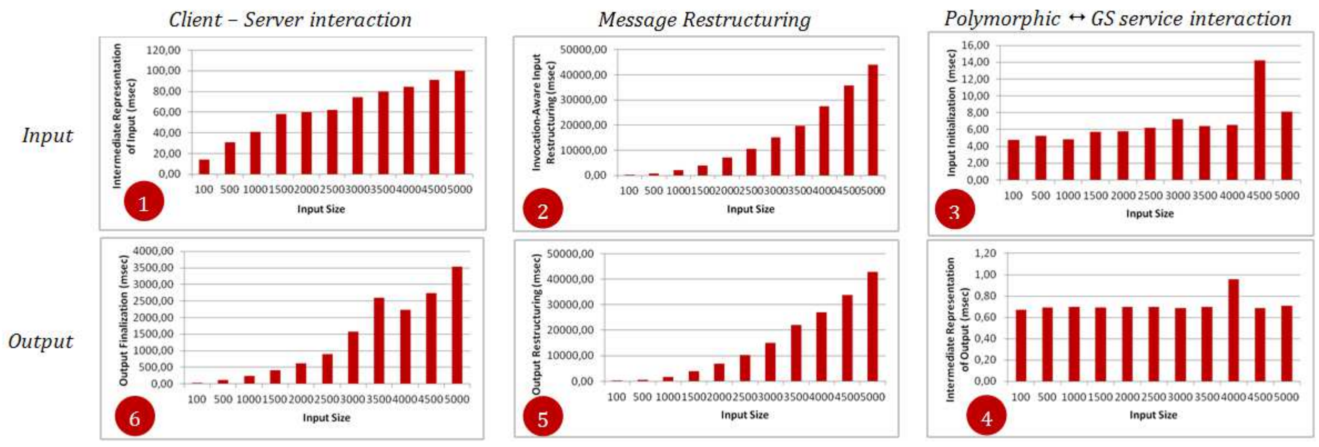


Figure 4.17: Time breakdown for the delegation overhead of the `AbstractWeatherForecastService`, when called as a Java API (numbers indicate the different phases of the translation of abstract to concrete invocations)

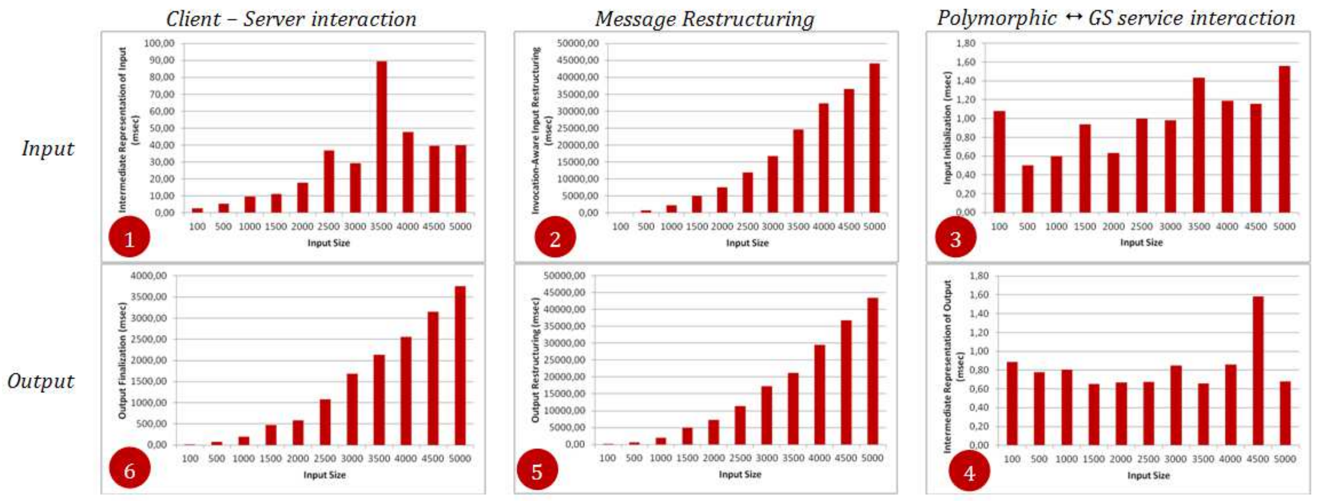


Figure 4.18: Time breakdown for the delegation overhead of the `AbstractWeatherForecastService`, when called as a Web Service (numbers indicate the different phases of the translation of abstract to concrete invocations)

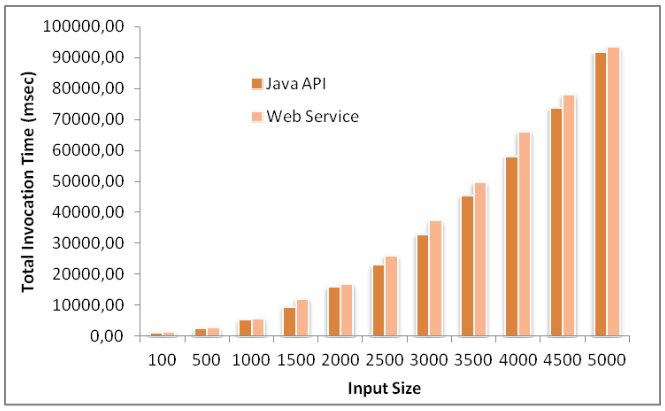


Figure 4.19: Total execution time for the invocation of the `AbstractWeatherForecastService`

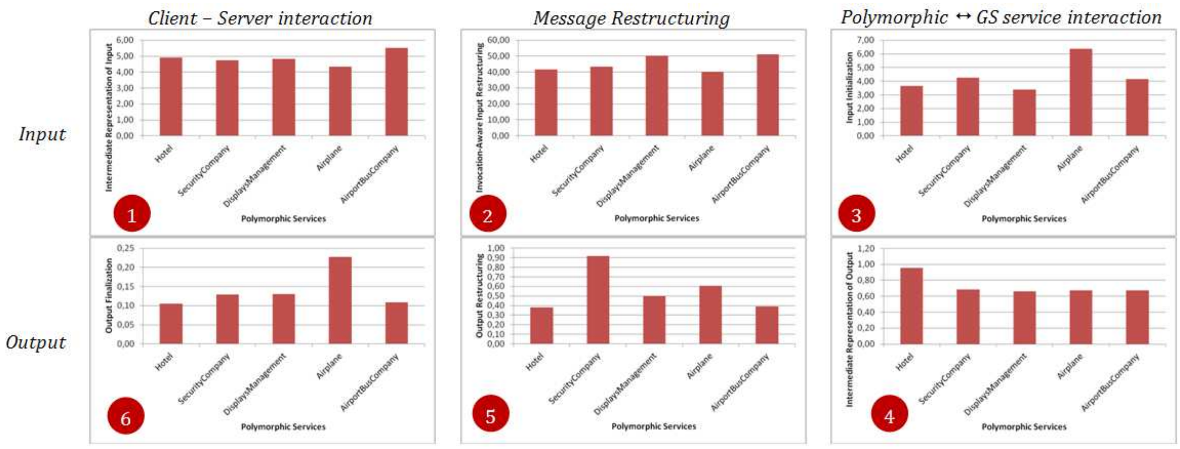


Figure 4.20: Delegation overhead breakdown for `AbstractHotel`, `AbstractSecurityCompany`, `AbstractAirplane`,`AbstractDisplaysManagement`, and `AbstractAirportBusCompany`, when called as Java APIs (numbers indicate the different phases of the translation of abstract to concrete invocations)

- The time needed for the notification of the services that use the substituted service for the completion of the substitution process, increases with the number of these services, for reasons similar to the above observation.
- The time needed for changing the mapping that is used by the functional abstraction service is independent of the number of services that use the substituted service.
- The total time needed for the service substitution increases in accordance with the number of services that use the substituted service. This is due to the two notification phases, as already explained.

CONCLUSION: To sum up, there are two main take-away messages from the experimental evaluation of the abstraction-oriented service substitution: (1) service substitution introduces an execution overhead that is proportional to the amount of data that is transferred via functional abstraction services; (2) supporting consistent substitution by notifying affected services further introduces an execution overhead that is proportional to the number of affected services. Overall, as anticipated extensibility and adaptability have a negative effect on performance. The use of middleware that allows to achieve these properties should be conservative, i.e., it should be used if these properties are really necessary.

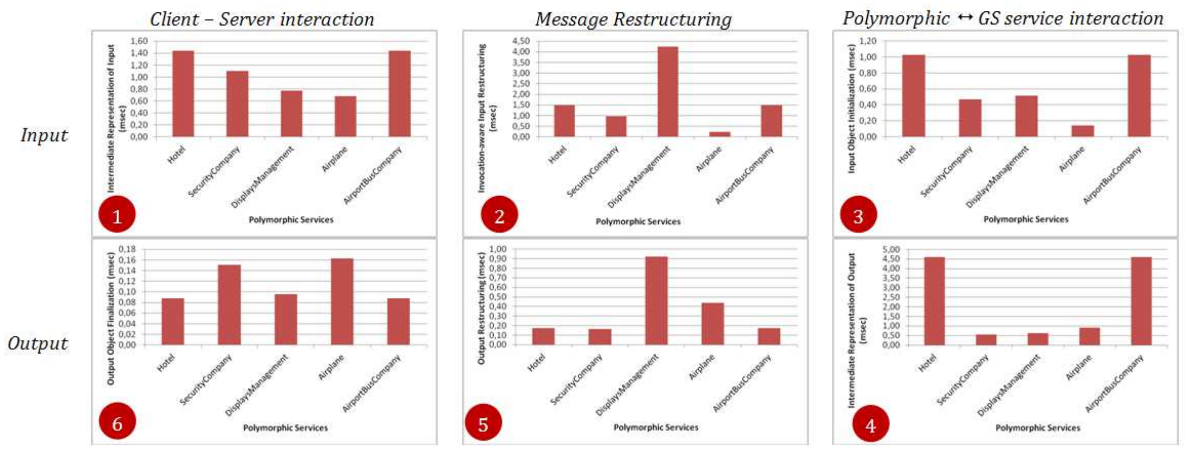


Figure 4.21: Delegation overhead breakdown for `AbstractHotel`, `AbstractSecurityCompany`, `AbstractAirplane`,`AbstractDisplaysManagement`, and `AbstractAirportBusCompany`, when called as Web services (numbers indicate the different phases of the translation of abstract to concrete invocations)

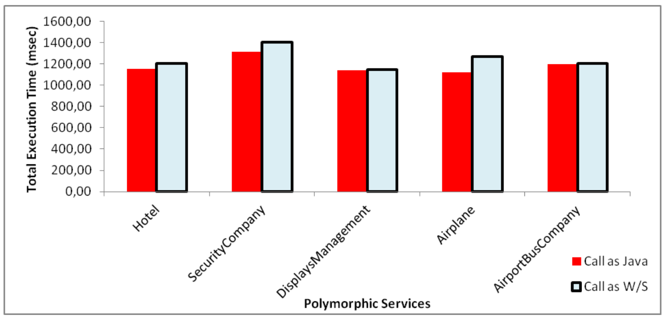


Figure 4.22: Total execution time for `AbstractHotel`, `AbstractSecurityCompany`, `AbstractAirplane`,`AbstractDisplaysManagement`, and `AbstractAirportBusCompany`

4.2.6. Cloud Enactment Engine Evaluation

Experimental Setup

We conducted experiments to evaluate the performance and scalability of the proposed Enactment Engine in terms of its capability to deploy a significant number of compositions onto a real-world cloud computing platform.

Our experiments use a synthetic choreography workload modeled as shown in Figure 4.24. The arrow direction is from the requester to the requested service. Although replies are not drawn for simplicity reasons, they are always sent back in a synchronous manner. This topology was chosen because (1) it contains most types of configurations found in choreographies, including branches and subsequent joints and (2) it follows a repetitive pattern that can be used to smoothly increase the size of the composition to analyze how the performance of the Enactment Engine behaves as its workload increases.

Initially, we conducted a multi-variable analysis of Enactment Engine by deploying service compositions in the following scenarios: 1) a small set of small compositions; 2) a small set of large compositions; 3) a large set of small compositions; 4) a larger ratio of services per node. Table 4.10 quantifies these scenarios.

In our experiments, the node allocation policy was the “limited round robin”, in which services are distributed across the available nodes, and the quantity of nodes is configured before each experiment. If the amount of services is not divisible by the number of nodes, some of the nodes will host one ad-

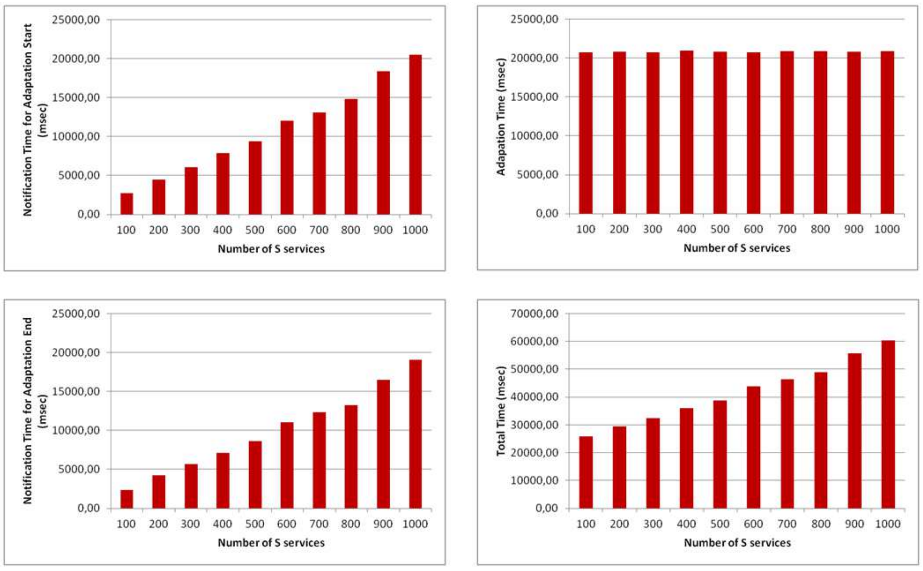


Figure 4.23: Time breakdown for the substituion overhead as a function of the number of services that use the substituted service

Table 4.10: Enactment Engine deployment scenarios

Scenario #	*# of Compositions*	*Size of each Composition*	*# of Cloud Nodes*	*# of Services per Node*
1	10	10	9	11 or 12
2	10	100	90	11 or 12
3	100	10	90	11 or 12
4	10	10	5	20

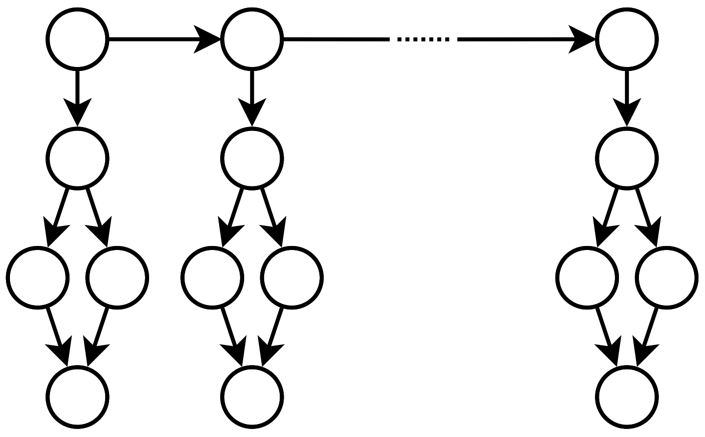


Figure 4.24: Choreography synthetic topology

Table 4.11: Choreography enactment time

Scenario	*Deployment time (s)*	*Successful Compositions*	*Successful Services*
1	467.9 ± 34.8	10.0 (100%) ± 0	100.0 (100%) ± 0
2	1477.1 ± 130.0	9.3 (93%) ± 0.3	999.3 (99.9%) ± 0.4
3	1455.2 ± 159.1	98.9 (98.9%) ± 0.8	998.5 (99.8%) ± 1.3
4	585.2 ± 38.1	10.0 (100%) ± 0.1	100.0 (100%) ± 0.1

ditional service. The idle nodes reserve size was five and the node creation timeout was 300 seconds. We used Amazon EC2 as the cloud computing service and the virtual machines were EC2 small instances, each one with 1.7 GiB of RAM, one vCPU with processing power equivalent to 1.0–1.2 GHz, and Ubuntu GNU/Linux 12.04. The Enactment Engine was executed on a machine with 8 GB of RAM, an Intel Core i7 CPU with 2.7 GHz and GNU/Linux kernel 3.6.7.

Each scenario was executed 30 times and all the values in Table 4.11 presents the average execution time with a confidence interval of 95%. It presents, for each scenario, the time necessary to deploy all the compositions plus the time to invoke them. It also shows how many compositions, as well as services, were successfully deployed for each scenario, for all executions.

We have also conducted a scalability experiment by varying in 4 steps the deployed choreography size and the amount of nodes available on the cloud environment, whereas keeping constant the number of deployed choreographies (only one) and the ratio of 10 deployed services per virtual machine. Each one of the 4 steps were executed 5 times. The environment used to run the Enactment Engine was a virtual machine (8GB RAM and 4 VCPU) running on top of a Open Stack installation. As in the previous experiments, the created nodes were Amazon EC2 instances. The results of this experiment are shown on Table 4.12, where for each scenario we have the average deployment time and the average number of successfully deployed services, both followed by theirs respective standard deviations.

Results & Findings

The results on Table 4.11 show that the Enactment Engine scales well in terms of the number of services being deployed. Although the number of services was multiplied by 10, the deployed time increased approximately only 3 times in scenarios 2 and 3. This time increment was caused mainly because the higher the number of services, the higher the likelihood of a fault that triggers the Enactment Engine fault-recovery mechanism, causing the re-execution of one or more tasks.

The results also show that when the number of services per node was doubled (scenario 4), the deployment time increased nearly 25%. Part of this overhead was caused by the increase on the

Table 4.12: Enactment Engine scalability analysis

Size of each composition	*Number of available nodes*	*Deployment time (s)*	*Successful Services*
200	10	623 ± 92	199.8 (99.9%) ± 0.4
600	30	1170 ± 260	599.2 (99.9%) ± 0.8
1000	50	1467 ± 140	998.4 (99.8%) ± 1.1
1400	70	1910 ± 374	1390.6 (99.3%)± 11.5

number of Chef scripts that must be executed (sequentially) on the nodes to reconfigure them.

During our experiments, we observed that the amount of failures was low: all the services were successfully deployed in more than 75% of the executions. By one failure, we mean that one service was not properly deployed due to problems discussed in section 2.4. Considering all the 30 executions, we got no failures in scenario 1, whereas we had only one failure in scenario 4. In scenario 2, the worst execution had 3 of 1,000 services not successfully deployed. In scenario 3, we got an execution with 20 failures, but it was an exceptional event, since the second worst situation had only 3 failures.

Regarding the reserve of idle nodes, as explained in section 2.4, we observed that 80% of the executions did not use it. When the node reserve was used, there was a maximum of six requests, but in most of the time there was only one. We also observed that the deployment time was not significantly affected when the (relatively frequent) failures on the cloud environment occurred, because new nodes were immediately retrieved from the reserve.

The results on Table 4.12 also show a good scalability in terms of deployed services. Increasing the number of deployed services in 5 times, the deployment time nearly increases 2.4 times, whereas increasing the number of deployed services in 7 times, the deployment time nearly increases 3.1 times. In absolute numbers, each increase in 200 deployed services was responsible for increasing the deployment time from 300 to 550 seconds.

CONCLUSION: The Enactment Engine scales well for different cases of choreography deployment, behaving well both with a large number of small choreographies and with a small number of large choreographies. In all cases, most of the time is spent by the underlying Cloud infrastructure to provide the nodes to host the services. The failures of the Amazon EC2 infrastructure are frequent and a special mechanism was inserted into the Enactment Engine to cope with them in a transparent way to the user.

The concurrent deployment of a large number of large choreographies (e.g., simultaneously deploying 100 choreographies of 100 services each) is not advisable because it would take too long to complete (possibly over an hour). Thus, choreography-based systems and applications should be designed with the idea that large choreographies are deployed not very often, but then used for a long time, e.g., a choreography may be executing for months.

We also consider that even the most higher deployed time (about 30 minutes for 1400 services) may be considered low if considering the long period a choreography should run and even considering the frequency of updates in choreographies, since both are much larger then 30 minutes.

5 Conclusion

The implementation of the CHOReOS Middleware has achieved a good level of maturity for an open source project and it is ready to be used in real-world, complex, distributed applications. Table 5.1 presents achievements, deficiencies, and needed enhancements of middleware components regarding Future Internet challenges. We hope the middleware components will continue to be developed after the end of the CHOReOS project and that further refinements, improvements, as well as new functionalities will be added.

The use of the middleware by the sub-teams working on the CHOReOS use-cases provided valuable feedback for the middleware development teams that were able to improve the implementation. The synthetic experiments and simulations performed help to give an idea of the current state of the implementation and provide hints on where the next optimization and enhancements should be made.

In the near future, the challenge will be to attract a community of developers and users to further continue the development of the system, which, we believe, can be very valuable for the next generation of computer systems for the Future Internet.

In the table below, we summarize the major benefits (Pros), limitations (Cons), and possibilities for future enhancements for each middleware component.

Middleware Component	Pros	Cons	Future Enhancements
Things C&E	Semantics-based automated service composition: Complex physical properties are derived from simpler ones.	Returns all possible compositions, which might be too many.	Optimizations to return the best set of services to be composed for a given request.
Service Substitution	Enables service substitution addressing the FI adaptation issue	FI adaptation further involves issues like recomposition, compensation, and re-negotiation.	Reducing the overhead of the substitution mechanism, along with combining the proposed approach with scalable recomposition, compensation, and re-negotiation.
XSB	Interconnection of services employing heterogeneous interaction paradigms; deployable on top of different communication substrates; extensibility to support new service middleware or new interaction paradigms.	In current prototype, throughput and latency values are satisfactory for relatively low numbers of concurrent service clients.	Optimizations for improving throughput and latency values in the case of many concurrent service clients; empirical evaluation of the framework facilities by different developers; extend with support for data streaming protocols.

Middleware Component	Pros	Cons	Future Enhancements
EasyESB	Efficient integration of multi-protocol technologies; compliance with cloud infrastructures and ability for a distributed deployment; efficient support for the coordination delegates for enacting the choreographies; ability of profiling EasyESB into a monitoring infrastructure for controlling services widely distributed.	EasyESB is a research prototype and thus needs maintenance and debugging.	Optimizations in resource usage; extending the cloud appliance to other solutions and improving the nodes automatic deployment.
LSB	Universal access to heterogeneous things-based services.	Dependence on proxy for 3G access may lead to issues at ultra large scale.	Investigate proxy-less solutions to access of thing-based services.
Service Discovery	Enables efficient service lookup over service abstractions, addressing the FI scale issue.	Service registration becomes more complex due to the extraction of service abstractions.	Selecting a small number of "important" service abstractions and making the extraction of abstractions semantic-aware and more interactive, without compromising performance.
Things Discovery	Scalable discovery thanks to the use of probabilistic approaches.	Fine-grained location data of services stored in registry, thus possibly leading to privacy issues.	Use only coarse-grained location data during registration, and fine-grained location data of services only during the lookup and access phases.
Grid as a Service	Better performance for CPU- and data-intensive tasks.	Requires a working grid installation.	Automate Grid installation writing Chef cookbooks for the Enactment Engine Middleware component.
Enactment Engine	Helps choreography developers in tackling scalability by enabling a fully automated deployment process; copes with technological heterogeneity by extension mechanisms; provides some mechanisms to enable cross-organizational choreographies; copes with adaptability by deploying monitoring infrastructure and providing means to service replication and service migration.	Does not tackle security issues; imposes some restrictions and obligations on services developers (although most of them may be also handled by EE extension).	Improving support to cross-organizational choreographies by federating different EE instances; better integrating the monitoring infrastructure with services scale up/down mechanisms; support to more technologies and cloud infrastructures.

Table 5.1: CHOReOS middleware components summary of achievements

Bibliography

[1] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Jist: an efficient approach to simulation using virtual machines: Research articles. *Softw. Pract. Exper.*, 35(6):539–576, May 2005.

[2] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz. Sumo - simulation of urban mobility: An overview. In *SIMUL 2011, The Third International Conference on Advances in System Simulation*, Barcelona, Spain, 2011.

[3] C. Bettstetter, G. Resta, and P. Santi. The node distribution of the random waypoint mobility model for wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, 2(3):257 – 269, july-sept. 2003.

[4] Thomas Bonald and James W. Roberts. Internet and the erlang formula. *SIGCOMM Comput. Commun. Rev.*, 42(1):23–30, January 2012.

[5] Ioannis Chatzigiannakis, Christos Koninis, Georgios Mylonas, Stefan Fischer, and Dennis Pfisterer. WISEBED: an open large-scale wireless sensor network testbed. In *Proceedings of the 1st International Conference on Sensor Networks Applications, Experimentation and Logistics*, September 2009.

[6] CHOReOS Project Team. D1.3: Choreos architectural style. Technical report, April 2011. `www.choreos.eu`.

[7] CHOReOS Project Team. D2.1: Choreos dynamic development model definition. Technical report, April 2011. `www.choreos.eu`.

[8] CHOReOS Project Team. D3.1: Choreos middleware specification. Technical report, April 2011. `www.choreos.eu`.

[9] CHOReOS Project Team. D1.4: Description of the choreos conceptual model and architectural style and their relation with the choreos development process and related methods, tools and middleware. Technical report, April 2012. `www.choreos.eu`.

[10] CHOReOS Project Team. D2.2: Definition of the dynamic development process for adaptable qos-aware uls choreographies. Technical report, April 2012. `www.choreos.eu`.

[11] CHOReOS Project Team. D3.2.2: Choreos middleware implementation. Technical report, October 2012. `www.choreos.eu`.

[12] CHOReOS Project Team. D4.2.1: Governance and v&v framework. Technical report, July 2012. `www.choreos.eu`.

[13] CHOReOS Project Team. D4.2.2: Governance and v&v framework. Technical report, October 2012. `www.choreos.eu`.

[14] CHOReOS Project Team. D5.3.2: Choreos idre and user manual revised version. Technical report, October 2012. `www.choreos.eu`.

[15] CHOReOS Project Team. D2.3: Choreos dynamic development process: methods and tools. Technical report, Octomber 2013. www.choreos.eu.

[16] CHOReOS Project Team. D4.3: Final release of the v&v tools and infrastructure. Technical report, April 2013. www.choreos.eu.

[17] CHOReOS Project Team. D9.6.2: Choreos courseware. Technical report, October 2013. www.choreos.eu.

[18] CONNECT Project Team. D1.2: Intermediate connect architecture. Technical report, February 2011. CONNECT ICT FET IP Project, http://connect-forever.eu.

[19] Clément Burin des Roziers, Guillaume Chelius, Tony Ducrocq, Eric Fleury, Antoine Fraboulet, Antoine Gallais, Nathalie Mitton, Thomas Noël, and Julien Vandaele. Using senslab as a first class scientific tool for large scale wireless sensor network experiments. In *Proceedings of the 10th international IFIP TC 6 conference on Networking - Volume Part I*, NETWORKING'11, pages 147–159, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Stein Desmet, Bruno Volckaert, Steven Van Assche, Bart Dhoedt, Filip De Turck, et al. Throughput evaluation of different enterprise service bus approaches. 2007.

[21] Nikolaos Georgantas, Georgios Bouloukakis, Sandrine Beauche, and Valérie Issarny. Service-oriented Distributed Applications in the Future Internet: The Case for Interaction Paradigm Interoperability. In Springer, editor, *ESOCC 2013 - European Conference on Service-Oriented and Cloud Computing*, Malaga, Spain, July 2013.

[22] J. Härri, F. Filali, C. Bonnet, and Marco Fiore. Vanetmobisim: generating realistic mobility patterns for vanets. In *Proceedings of the 3rd international workshop on Vehicular ad hoc networks*, VANET '06, pages 96–97, New York, NY, USA, 2006. ACM.

[23] Information Sciences Institute. The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[24] Haifeng Jiang, Howard Ho, Lucian Popa, and Wook-Shin Han. Mapping-driven xml transformation. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 1063–1072, 2007.

[25] State University of New York at Buffalo. PhoneLab. http://www.phone-lab.org.

[26] OMG. Deployment and configuration of component-based distributed applications (DEPL), April 2006. http://www.omg.org/spec/DEPL/.

[27] L. Sanchez, J.A. Galache, V. Gutierrez, J.M. Hernandez, J. Bernat, A. Gluhak, and T. Garcia. Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In *Future Network Mobile Summit (FutureNetw), 2011*, pages 1–8, 2011.

[28] Ken Ueno and Michiaki Tatsubori. Early capacity testing of an enterprise service bus. In *Web Services, 2006. ICWS'06. International Conference on Web Services*, pages 709–716. IEEE, 2006.

[29] Darthmouth University. CRAWDAD. http://crawdad.cs.dartmouth.edu.

[30] Ohio State University. KanseiGenie. http://kansei.cse.ohio-state.edu/KanseiGenie/.

[31] Yonsei University. LifeMap. http://lifemap.yonsei.ac.kr.

[32] Sandesh Uppoor, Oscar Trullols-Cruces, Marco Fiore, and Jose M. Barcelo-Ordinas. Generation and analysis of a large-scale urban vehicular mobility dataset. *IEEE Transactions on Mobile Computing*, 99(PrePrints):1, 2013.

[33] Andras Varga. The OMNeT++ Discrete Event Simulation System. In *European Simulation Multi-conference*, pages 319–324, Prague, Czech Republic, June 2001.

A Enactment Engine Listings

The listings cited on Section 4.1.5, Cloud Enactment Engine Use-Case-Based Evaluation, are the following.

```java
package org.ow2.choreos;

import java.util.Collections;

import org.ow2.choreos.chors.datamodel.ChoreographySpec;
import org.ow2.choreos.nodes.datamodel.ResourceImpact;
import org.ow2.choreos.services.datamodel.DeployableServiceSpec;
import org.ow2.choreos.services.datamodel.PackageType;
import org.ow2.choreos.services.datamodel.ServiceDependency;
import org.ow2.choreos.services.datamodel.ServiceType;

public class ThalesSpecs {

    public static final String AIRPORT = "airport";
    public static final String AIRPORT_BUS_COMPANY = "airportbuscompany";
    public static final String AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR = "airportdisplayactuatorsaggregator";
    public static final String AIRPORT_INFRARED_SENSORS_AGGREGATOR = "airportinfraredsensorsaggregator";
    public static final String AIRPORT_NOISE_SENSORS_AGGREGATOR = "aipornoisesensorsaggregator";
    public static final String AIRPORT_PRESSURE_SENSORS_AGGREGATOR = "aiportpressuresensorsaggregator";
    public static final String AIRPORT_SIGN_ACTUATORS_AGGREGATOR = "airportsignactuatorsaggregator";
    public static final String AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR = "airportspeakeractuatorsaggregator";
    public static final String BOOKABLE_AMENITY = "bookableamenity";
    public static final String LUGGAGE_HANDLING_COMPANY = "luggagehandlingcompany";
    public static final String MID_DISPLAY_ACTUATORS_AGGREGATOR = "middisplayactuatorsaggregator";
    public static final String MID_LOCATION_SENSORS_AGGREGATOR = "midlocationsensorsaggregator";
    public static final String MID_MICROPHONE_SENSORS_AGGREGATOR = "midmicrophonesensorsaggregator";
    public static final String SECURITY_COMPANY = "securitycompany";
    public static final String STAND_AND_GATE_MANAGEMENT = "standandgatemanagement";

    public static final String AIRPORT_JAR = "http://sd-49168.dedibox.fr/DeployableServices/airport-service.jar";
    public static final String AIRPORT_BUS_COMPANY_JAR = "http://sd-49168.dedibox.fr/DeployableServices/airportbuscompany-service.jar";
    public static final String AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/airportdisplayactuatorsaggregator-service.jar";
    public static final String AIRPORT_INFRARED_SENSORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/airportinfraredsensorsaggregator-service.jar";
    public static final String AIRPORT_NOISE_SENSORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/airportnoisesensorsaggregator-service.jar";
    public static final String AIRPORT_PRESSURE_SENSORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/airportpressuresensorsaggregator-service.jar";
    public static final String AIRPORT_SIGN_ACTUATORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/airportsignactuatorsaggregator-service.jar";
    public static final String AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/airportspeakeractuatorsaggregator-service.jar";
    public static final String BOOKABLE_AMENITY_JAR = "http://sd-49168.dedibox.fr/DeployableServices/bookableamenity-service.jar";
    public static final String LUGGAGE_HANDLING_COMPANY_JAR = "http://sd-49168.dedibox.fr/DeployableServices/luggagehandlingcompany-service.jar";
    public static final String MID_DISPLAY_ACTUATORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/middisplayactuatorsaggregator-service.jar";
    public static final String MID_LOCATION_SENSORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/midlocationsensorsaggregator-service.jar";
    public static final String MID_MICROPHONE_SENSORS_AGGREGATOR_JAR = "http://sd-49168.dedibox.fr/DeployableServices/midmicrophonesensorsaggregator-service.jar";
    public static final String SECURITY_COMPANY_JAR = "http://sd-49168.dedibox.fr/DeployableServices/securitycompany-service.jar";
    public static final String STAND_AND_GATE_MANAGEMENT_JAR = "http://sd-49168.dedibox.fr/DeployableServices/standandgatemanagement-service.jar";

    public static final int AIRPORT_PORT = 8004;
    public static final int AIRPORT_BUS_COMPANY_PORT = 8023;
    public static final int AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR_PORT = 8006;
    public static final int AIRPORT_INFRARED_SENSORS_AGGREGATOR_PORT = 8007;
    public static final int AIRPORT_NOISE_SENSORS_AGGREGATOR_PORT = 8008;
    public static final int AIRPORT_PRESSURE_SENSORS_AGGREGATOR_PORT = 8024;
    public static final int AIRPORT_SIGN_ACTUATORS_AGGREGATOR_PORT = 8010;
    public static final int AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR_PORT = 8011;
    public static final int BOOKABLE_AMENITY_PORT = 8013;
    public static final int LUGGAGE_HANDLING_COMPANY_PORT = 8017;
    public static final int MID_DISPLAY_ACTUATORS_AGGREGATOR_PORT = 8018;
    public static final int MID_LOCATION_SENSORS_AGGREGATOR_PORT = 8019;
    public static final int MID_MICROPHONE_SENSORS_AGGREGATOR_PORT = 8020;
    public static final int SECURITY_COMPANY_PORT = 8021;
    public static final int STAND_AND_GATE_MANAGEMENT_PORT = 8022;

    private final ResourceImpact resourceImpact = new ResourceImpact();

    private final String serviceVersion = "0.1";

    private ChoreographySpec chorSpec;

    private DeployableServiceSpec airportSpec;
    private DeployableServiceSpec airportBusCompanySpec;
    private DeployableServiceSpec airportDisplayActuatorsAggregatorSpec;
    private DeployableServiceSpec airportInfraredSensorsAggregatorSpec;
    private DeployableServiceSpec airportNoiseSensorsAggregatorSpec;
    private DeployableServiceSpec airportPressureSensorsAggregatorSpec;
    private DeployableServiceSpec airportSignActuatorsAggregatorSpec;
    private DeployableServiceSpec airportSpeakerActuatorsAggregatorSpec;
    private DeployableServiceSpec bookableAmenitySpec;
    private DeployableServiceSpec luggageHandlingCompanySpec;
    private DeployableServiceSpec midDisplayActuatorsAggregatorSpec;
    private DeployableServiceSpec midLocationSensorsAggregatorSpec;
    private DeployableServiceSpec midMicrophoneSensorsAggregatorSpec;
    private DeployableServiceSpec securityCompanySpec;
    private DeployableServiceSpec standAndGateManagementSpec;

    public ThalesSpecs() {
        initAirportSpecs();
        initAirportBusCompanySpecs();
        initAirportDisplayActuatorsAggregatorSpecs();
        initAirportInfraredSensorsAggregatorSpecs();
        initAirportNoiseSensorsAggregatorSpecs();
        initAirportPressureSensorsAggregatorSpecs();
        initAirportSignActuatorsAggregatorSpecs();
        initAirportSpeakerActuatorsAggregatorSpecs();
        initBookableAmenitySpecs();
        initLuggageHandlingCompanySpecs();
        initMIDDisplayActuatorsAggregatorSpecs();
        initMIDLocationSensorsAggregatorSpecs();
        initMIDMicrophoneSensorsAggregatorSpecs();
        initsecurityCompanySpecs();
        initStandAndGateManagementSpecs();
        createChorSpec();
    }

    private void createChorSpec() {
        this.chorSpec = new ChoreographySpec(this.airportSpec, this.airportBusCompanySpec,
                this.airportDisplayActuatorsAggregatorSpec, this.airportInfraredSensorsAggregatorSpec,
                this.airportNoiseSensorsAggregatorSpec, this.airportPressureSensorsAggregatorSpec,
                this.airportSignActuatorsAggregatorSpec, this.airportSpeakerActuatorsAggregatorSpec,
                this.bookableAmenitySpec, this.luggageHandlingCompanySpec, this.midDisplayActuatorsAggregatorSpec,
                this.midLocationSensorsAggregatorSpec, this.midMicrophoneSensorsAggregatorSpec,
                this.securityCompanySpec, this.standAndGateManagementSpec);
    }

    private void initStandAndGateManagementSpecs() {
        standAndGateManagementSpec = new DeployableServiceSpec(STAND_AND_GATE_MANAGEMENT, ServiceType.SOAP,
                PackageType.COMMAND_LINE, resourceImpact, serviceVersion, STAND_AND_GATE_MANAGEMENT_JAR,
                STAND_AND_GATE_MANAGEMENT_PORT, STAND_AND_GATE_MANAGEMENT, 1);
        standAndGateManagementSpec.setRoles(Collections.singletonList(STAND_AND_GATE_MANAGEMENT));
        standAndGateManagementSpec.addDependency(new ServiceDependency(AIRPORT, AIRPORT));
    }

    private void initsecurityCompanySpecs() {
        securityCompanySpec = new DeployableServiceSpec(SECURITY_COMPANY, ServiceType.SOAP, PackageType.COMMAND_LINE,
                resourceImpact, serviceVersion, SECURITY_COMPANY_JAR, SECURITY_COMPANY_PORT, SECURITY_COMPANY, 1);
        securityCompanySpec.setRoles(Collections.singletonList(SECURITY_COMPANY));
        securityCompanySpec.addDependency(new ServiceDependency(AIRPORT, AIRPORT));
    }

    private void initMIDMicrophoneSensorsAggregatorSpecs() {
        midMicrophoneSensorsAggregatorSpec = new DeployableServiceSpec(MID_MICROPHONE_SENSORS_AGGREGATOR,
                ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact, serviceVersion,
                MID_MICROPHONE_SENSORS_AGGREGATOR_JAR, MID_MICROPHONE_SENSORS_AGGREGATOR_PORT,
                MID_MICROPHONE_SENSORS_AGGREGATOR, 1);
        midMicrophoneSensorsAggregatorSpec.setRoles(Collections.singletonList(MID_MICROPHONE_SENSORS_AGGREGATOR));
    }

    private void initMIDLocationSensorsAggregatorSpecs() {
        midLocationSensorsAggregatorSpec = new DeployableServiceSpec(MID_LOCATION_SENSORS_AGGREGATOR, ServiceType.SOAP,
                PackageType.COMMAND_LINE, resourceImpact, serviceVersion, MID_LOCATION_SENSORS_AGGREGATOR_JAR,
                MID_LOCATION_SENSORS_AGGREGATOR_PORT, MID_LOCATION_SENSORS_AGGREGATOR, 1);
        midLocationSensorsAggregatorSpec.setRoles(Collections.singletonList(MID_LOCATION_SENSORS_AGGREGATOR));
    }

    private void initMIDDisplayActuatorsAggregatorSpecs() {
        midDisplayActuatorsAggregatorSpec = new DeployableServiceSpec(MID_DISPLAY_ACTUATORS_AGGREGATOR,
                ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact, serviceVersion,
                MID_DISPLAY_ACTUATORS_AGGREGATOR_JAR, MID_DISPLAY_ACTUATORS_AGGREGATOR_PORT,
                MID_DISPLAY_ACTUATORS_AGGREGATOR, 1);
        midDisplayActuatorsAggregatorSpec.setRoles(Collections.singletonList(MID_DISPLAY_ACTUATORS_AGGREGATOR));
    }

    private void initLuggageHandlingCompanySpecs() {
        luggageHandlingCompanySpec = new DeployableServiceSpec(LUGGAGE_HANDLING_COMPANY, ServiceType.SOAP,
                PackageType.COMMAND_LINE, resourceImpact, serviceVersion, LUGGAGE_HANDLING_COMPANY_JAR,
                LUGGAGE_HANDLING_COMPANY_PORT, LUGGAGE_HANDLING_COMPANY, 1);
        luggageHandlingCompanySpec.setRoles(Collections.singletonList(LUGGAGE_HANDLING_COMPANY));
        luggageHandlingCompanySpec.addDependency(new ServiceDependency(AIRPORT, AIRPORT));
    }

    private void initBookableAmenitySpecs() {
        bookableAmenitySpec = new DeployableServiceSpec(BOOKABLE_AMENITY, ServiceType.SOAP, PackageType.COMMAND_LINE,
                resourceImpact, serviceVersion, BOOKABLE_AMENITY_JAR, BOOKABLE_AMENITY_PORT, BOOKABLE_AMENITY, 1);
        bookableAmenitySpec.setRoles(Collections.singletonList(BOOKABLE_AMENITY));
    }

    private void initAirportSpeakerActuatorsAggregatorSpecs() {
        airportSpeakerActuatorsAggregatorSpec = new DeployableServiceSpec(AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR,
                ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact, serviceVersion,
                AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR_JAR, AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR_PORT,
                AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR, 1);
        airportSpeakerActuatorsAggregatorSpec.setRoles(Collections.singletonList(AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR));
    }

    private void initAirportSignActuatorsAggregatorSpecs() {
        airportSignActuatorsAggregatorSpec = new DeployableServiceSpec(AIRPORT_SIGN_ACTUATORS_AGGREGATOR,
                ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact, serviceVersion,
                AIRPORT_SIGN_ACTUATORS_AGGREGATOR_JAR, AIRPORT_SIGN_ACTUATORS_AGGREGATOR_PORT,
                AIRPORT_SIGN_ACTUATORS_AGGREGATOR, 1);
        airportSignActuatorsAggregatorSpec.setRoles(Collections.singletonList(AIRPORT_SIGN_ACTUATORS_AGGREGATOR));
    }

    private void initAirportPressureSensorsAggregatorSpecs() {
        airportPressureSensorsAggregatorSpec = new DeployableServiceSpec(AIRPORT_PRESSURE_SENSORS_AGGREGATOR,
                ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact, serviceVersion,
                AIRPORT_PRESSURE_SENSORS_AGGREGATOR_JAR, AIRPORT_PRESSURE_SENSORS_AGGREGATOR_PORT,
                AIRPORT_PRESSURE_SENSORS_AGGREGATOR, 1);
        airportPressureSensorsAggregatorSpec.setRoles(Collections.singletonList(AIRPORT_PRESSURE_SENSORS_AGGREGATOR));
    }

    private void initAirportNoiseSensorsAggregatorSpecs() {
        airportNoiseSensorsAggregatorSpec = new DeployableServiceSpec(AIRPORT_NOISE_SENSORS_AGGREGATOR,
                ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact, serviceVersion,
                AIRPORT_NOISE_SENSORS_AGGREGATOR_JAR, AIRPORT_NOISE_SENSORS_AGGREGATOR_PORT,
                AIRPORT_NOISE_SENSORS_AGGREGATOR, 1);
        airportNoiseSensorsAggregatorSpec.setRoles(Collections.singletonList(AIRPORT_NOISE_SENSORS_AGGREGATOR));
    }

    private void initAirportInfraredSensorsAggregatorSpecs() {
        airportInfraredSensorsAggregatorSpec = new DeployableServiceSpec(AIRPORT_INFRARED_SENSORS_AGGREGATOR,
                ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact, serviceVersion,
                AIRPORT_INFRARED_SENSORS_AGGREGATOR_JAR, AIRPORT_INFRARED_SENSORS_AGGREGATOR_PORT,
                AIRPORT_INFRARED_SENSORS_AGGREGATOR, 1);
        airportInfraredSensorsAggregatorSpec.setRoles(Collections.singletonList(AIRPORT_INFRARED_SENSORS_AGGREGATOR));
    }

    private void initAirportDisplayActuatorsAggregatorSpecs() {
        airportDisplayActuatorsAggregatorSpec = new DeployableServiceSpec(AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR,
                ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact, serviceVersion,
                AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR_JAR, AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR_PORT,
                AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR, 1);
        airportDisplayActuatorsAggregatorSpec.setRoles(Collections.singletonList(AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR));
    }

    private void initAirportBusCompanySpecs() {
        airportBusCompanySpec = new DeployableServiceSpec(AIRPORT_BUS_COMPANY, ServiceType.SOAP,
                PackageType.COMMAND_LINE, resourceImpact, serviceVersion, AIRPORT_BUS_COMPANY_JAR,
                AIRPORT_BUS_COMPANY_PORT, AIRPORT_BUS_COMPANY, 1);
        airportBusCompanySpec.setRoles(Collections.singletonList(AIRPORT_BUS_COMPANY));
        airportBusCompanySpec.addDependency(new ServiceDependency(AIRPORT, AIRPORT));
    }

    private void initAirportSpecs() {
        airportSpec = new DeployableServiceSpec(AIRPORT, ServiceType.SOAP, PackageType.COMMAND_LINE, resourceImpact,
                serviceVersion, AIRPORT_JAR, AIRPORT_PORT, AIRPORT, 1);
        airportSpec.setRoles(Collections.singletonList(AIRPORT));
        airportSpec.addDependency(new ServiceDependency(AIRPORT_BUS_COMPANY, AIRPORT_BUS_COMPANY));
        airportSpec.addDependency(new ServiceDependency(AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR,
                AIRPORT_DISPLAY_ACTUATORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(AIRPORT_INFRARED_SENSORS_AGGREGATOR,
                AIRPORT_INFRARED_SENSORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(AIRPORT_NOISE_SENSORS_AGGREGATOR,
                AIRPORT_NOISE_SENSORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(AIRPORT_PRESSURE_SENSORS_AGGREGATOR,
                AIRPORT_PRESSURE_SENSORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(AIRPORT_SIGN_ACTUATORS_AGGREGATOR,
                AIRPORT_SIGN_ACTUATORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR,
                AIRPORT_SPEAKER_ACTUATORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(BOOKABLE_AMENITY, BOOKABLE_AMENITY));
        airportSpec.addDependency(new ServiceDependency(LUGGAGE_HANDLING_COMPANY, LUGGAGE_HANDLING_COMPANY));
        airportSpec.addDependency(new ServiceDependency(MID_DISPLAY_ACTUATORS_AGGREGATOR,
                MID_DISPLAY_ACTUATORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(MID_LOCATION_SENSORS_AGGREGATOR,
                MID_LOCATION_SENSORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(MID_MICROPHONE_SENSORS_AGGREGATOR,
                MID_MICROPHONE_SENSORS_AGGREGATOR));
        airportSpec.addDependency(new ServiceDependency(SECURITY_COMPANY, SECURITY_COMPANY));
        airportSpec.addDependency(new ServiceDependency(STAND_AND_GATE_MANAGEMENT, STAND_AND_GATE_MANAGEMENT));
    }

    public ChoreographySpec getChorSpec() {
        return chorSpec;
    }

}
```

Listing A.1: WP6 choreography specification used as Enactment Engine input

```java
package org.ow2.choreos;

import org.ow2.choreos.chors.ChoreographyDeployer;
import org.ow2.choreos.chors.ChoreographyNotFoundException;
import org.ow2.choreos.chors.EnactmentException;
import org.ow2.choreos.chors.client.ChorDeployerClient;
import org.ow2.choreos.chors.datamodel.Choreography;
import org.ow2.choreos.chors.datamodel.ChoreographySpec;
import org.ow2.choreos.utils.Alarm;
import org.ow2.choreos.utils.CommandLineException;

public class ThalesEnact {

    public static void main(String[] args) throws EnactmentException, ChoreographyNotFoundException, CommandLineException {

        final String CHOR_DEPLOYER_URI = "http://localhost:9102/choreographydeployer";
        ChoreographyDeployer chorDeployer = new ChorDeployerClient(CHOR_DEPLOYER_URI);
        ThalesSpecs thalesSpecs = new ThalesSpecs();
        ChoreographySpec chorSpec = thalesSpecs.getChorSpec();

        String chorId = chorDeployer.createChoreography(chorSpec);
        Choreography chor = chorDeployer.enactChoreography(chorId);

        System.out.println(chor); // just to check EE output
    }
}
```

Listing A.2: Program to invoke Enactment Engine and launch WP6 choreography deployment