

master's thesis

# Methods and Tools for Intelligent ESB

*Michal Fuksa*



May 2014

advisor: Ing.Vrba Pavel Ph.D.

Czech Technical University in Prague  
Faculty of Electrical Engineering, Department of Computer  
Science and Engineering

## Acknowledgement

I would like to express my appreciation and thanks to my advisor Ing.Vrba Pavel Ph.D. and Ing. Martin Klíma Ph.D for all the support and advisement. Big thanks go to my family for the support and friends for piece of mind.

I'd like to thank Ing. Tomáš Budín for the patience and all the help. Thanks goes to all my colleagues in Certicon and to the Certicon itself for chance to do this thesis.

I'd like to thank my university for all the wonderful people and experience that I've witnessed, and I would like to thank my country, Czech Republic, for allowing me to get out in the world with education and without debts, took me some time on travels to appreciate that.

To Lubomir Salek, much help, very thanks, wow.

Thanks Obama.

Czech Technical University in Prague  
Faculty of Electrical Engineering

Department of Computer Science and Engineering

## DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Michal Fuksa**

Study programme: Open Informatics  
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Methods and Tools for Intelligent ESB**

### Guidelines:

The goal of the work is to develop a set of methods and tools that support the implementation of Intelligent Enterprise Service Bus (iESB), which is created within the European project ARUM. The set will contain the following components:

- 1) Tool for remote deployment of services in the JBoss ESB
- 2) Software architecture for integration of multi-agent system (JADE platform) with JBoss ESB. Architecture will allow for communication between a JADE agent, wrapped in a ESB service with other ESB service or with another JADE agent running in another ESB service.
- 3) Database service provides other ESB services with the central database storage. The database service will support various types of databases (SQL/noSQL); provides access rights management, offers CRUD operations on database; and supports publish-subscribe mechanism for listening to data changes.

The work will be supplemented with the technical documentation.

### Bibliography/Sources:


F. Bellifemine, G. Caire, and D. Greenwood, Developing multi-agent systems with JADE. Chichester: Wiley, 2007.

C. Coronel, S. Morris, P. Rob, Database Systems: Design, Implementation, and Management. Course Technology, Cengage Learning, 2013

JBoss ESB Documentation. [Online]. Available: <http://www.jboss.org/jbossesb/docs/index>

Diploma Thesis Supervisor: Ing. Pavel Vrba, Ph.D.

Valid until the end of the summer semester of academic year 2014/2015

  
doc. Ing. Filip Železný, Ph.D.  
Head of Department



Prague, March 3, 2014

  
prof. Ing. Pavel Ripka, CSc.  
Dean

## Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

Prague 7.5.2014

  
.....

## Abstract

Softwarové systémy velkých společností, které zahrnují velké množství počítačů a umožňují spolupráci mnoha lidí, mohou růst do nevladatelných rozměrů. Údržba a rozšíření těchto systémů, postavených jako jediný softwarový monolit, se stává komplexní a zdroje potřebné pro práci s tímto systémem rostou značně super-lineárně. Aby tyto systémy byly jednodušší, lze je implementovat jako architekturu orientovanou na služby (SOA). Je to přístup, kdy jsou systémy rozděleny do několika samostatných subsystémů (služeb), jejichž cílem je řídit jedinou funkci celého systému.

Tato práce popisuje implementaci tří komponent pro Arum (adaptivní řízení výroby) projekt, který je založen na Enterprise Service Bus (ESB) implementaci SOA. Projekt Arum vytváří sofistikovaný softwarový systém, který bude schopen spravovat lidské a materiální zdroje, plánování, rozvrhování a dohled nad výrobou komerčního tryskového letadla.

Těmito třemi komponentami jsou zaprvé Nástroj na nasazování služeb (Deployer), používaný správci pro účinné nasazování dalších komponent na ESB, zadruhé knihovna Jade brány (JadeGateway) umožňující agentům Jade platformy komunikovat s ESB službami a naopak a zatřetí databázová služba (DatabaseService), která zapouzdřuje více databázových systémů, unifikuje přístup prostřednictvím společného protokolu a poskytuje zapouzdřené databáze jako službu.

## Klíčová slova

SOA; ESB; Java; Databáze; Jade . . .

## **Abstract**

Enterprise software systems, managing large numbers of computers and allowing numerous people to cooperate, can grow to unmanageable scales. Maintenance and extension of such systems constructed as single software monoliths gets complex and resources needed to work with such systems grow in a significantly superlinear fashion. In order to make those systems simpler, developers can adopt the Service oriented architecture (SOA) approach, where systems are divided into multiple separate subsystems, services, whose purpose it is to respectively manage a single function of the system.

This thesis describes the implementation of three components for the ARUM (Adaptive Production Management) project, which is based on Enterprise service bus (ESB) implementation of SOA. The ARUM project creates a highly sophisticated software system, that will be capable of managing human and material resources, planning, scheduling and supervising the manufacture of a commercial jet airliner.

These three components consist of firstly a deployment tool, used by administrators to effectively deploy and undeploy components to the ESB, secondly a Jade gateway library, allowing Jade platform agents to communicate with ESB services and vice versa and thirdly a database service that encapsulates multiple database engines and types and unifies the access through a common protocol and provides encapsulated databases as a service.

## **Keywords**

SOA; ESB; Java;Databases;Jade . . .

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Airplane manufacture . . . . .	2
1.1.1. Airplane construction . . . . .	2
1.1.2. Manufacture planning . . . . .	2
1.1.3. SOA integration . . . . .	3
1.1.4. ARUM . . . . .	3
1.2. ESB . . . . .	3
Message routing . . . . .	4
Service deployment and versioning . . . . .	4
Event handling, data transformations and other fancy functions . . . . .	4
1.3. ARUM iESB . . . . .	4
1.3.1. Other iESB contents . . . . .	6
Ontology service . . . . .	6
Scheduler . . . . .	6
Factory network designer . . . . .	6
Scenario designer . . . . .	6
Jade gateway library . . . . .	6
Database service . . . . .	6
<b>2. Deployer</b>	<b>7</b>
2.1. Analysis . . . . .	7
2.2. ARUMPackage . . . . .	7
2.3. Deployment process . . . . .	8
2.3.1. ARUMPackage to XML . . . . .	8
2.4. DeployerPortlet and ARUMAdminService . . . . .	8
Configuration . . . . .	9
2.5. Deployer service . . . . .	10
2.5.1. Simple message based file transfer . . . . .	10
2.6. Under development . . . . .	10
2.6.1. Portlet . . . . .	10
2.6.2. Extension of functionality . . . . .	11
<b>3. Jade gateway</b>	<b>12</b>
3.1. Analysis . . . . .	12
3.1.1. MAS vs ESB . . . . .	12
3.1.2. Jade . . . . .	12
3.1.3. Jade vs. iESB . . . . .	13
3.2. Implementation . . . . .	13
3.2.1. Architecture . . . . .	13
3.2.2. Gateway . . . . .	15
Routing . . . . .	16
Configuration of the gateway . . . . .	17
Communication table . . . . .	17
3.2.3. Gateway agent . . . . .	18
3.2.4. Communication protocols . . . . .	19
Conversion . . . . .	19
Receive communicationID protocol . . . . .	20

3.2.5.	Synchronous to Asynchronous conversion . . . . .	20
3.2.6.	User API . . . . .	22
	Agent-aware service . . . . .	22
	ESB-aware agent . . . . .	24
	ESBAwareSideContainer . . . . .	25
3.2.7.	Interfaces and classes . . . . .	25
	NonMasReceiver . . . . .	25
	ExternalAgentContainer . . . . .	25
	Package examples . . . . .	25
3.2.8.	Future upgrades . . . . .	26
<b>4.</b>	<b>Database service</b>	<b>27</b>
4.1.	Analysis . . . . .	27
	4.1.1. Use cases . . . . .	27
4.2.	Architecture . . . . .	28
	4.2.1. Pluggable databases . . . . .	30
	4.2.2. Dataspaces . . . . .	31
	4.2.3. Authorization . . . . .	32
	4.2.4. Listeners . . . . .	33
	4.2.5. Database types . . . . .	33
	Key-value dataspace type . . . . .	34
	Column dataspace type . . . . .	35
4.3.	XML serialization . . . . .	36
	4.3.1. Database event listening . . . . .	36
	4.3.2. Database schema . . . . .	37
	4.3.3. Authorization . . . . .	37
	4.3.4. Dataspace . . . . .	37
4.4.	Protocols . . . . .	38
	4.4.1. Data manipulation protocol . . . . .	38
	4.4.2. Dataspace administration protocol . . . . .	41
	4.4.3. ADS lifecycle . . . . .	43
	Startup . . . . .	43
	New dataspace backup . . . . .	44
4.5.	Deployment . . . . .	44
4.6.	Other interfaces and classes . . . . .	45
4.7.	Future work . . . . .	46
<b>5.</b>	<b>Conclusions</b>	<b>47</b>
<b>Appendices</b>		
<b>A.</b>	<b>Appendix</b>	<b>48</b>
	A.1. ARUM database service XML/XSD examples . . . . .	48
	A.1.1. Data manipulation XML . . . . .	48
<b>B.</b>	<b>CD contents</b>	<b>51</b>



## Abbreviations

This section contains frequently used abbreviations used throughout this thesis.

abbrv.	explanation
ADS	ARUM Database Service
Agent	Autonomous piece of program capable of sending and receiving messages and reacting on them.
ARUM	Adaptive production management. Project developing SOA based software for control of ramp-up manufacture.
CLP	Constraint Logic Programming.
CRUD	Create/Read/Update/Delete basic storage operations.
ESB	Enterprise service bus. Implementation of SOA.
FIPA	Foundation for Intelligent Physical Agents. IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies.[1]
GUI	Graphic user interface.
iESB	Intelligent Enterprise service bus. ARUM implementation of SOA and extension of ESB.
iESB	Intelligent ESB. Product of ARUM project.
JADE	Java Agent DEvelopment Framework. Platform for development of MAS.
JAR	Java executable ARchive.
JAXB	Java Architecture for XML Binding
JG	JADE gateway
MAS	Multi agent system. Software system consisted of set of agents capable of computation and communication through the platform.
noSQL	Not only SQL. Set of database format that specialize in a specific data type.
RCID	Receive CommunicationID protocol. Protocol used in communication from JADE platform to iESB.
SOA	Service oriented architecture. Implementation style.
SQL	Structured Query Language. Language used to access databases.
XML	Extensible Markup Language
XSD	XML Schema Definition

# 1. Introduction

Service oriented architecture (SOA) is rightfully receiving a lot of attention for its flexibility, modularity and simplicity. It is being used especially in large corporate installations, where its modularity really comes in handy.

Adaptive Production Management (ARUM) is a large scale production management software system that will be used for control of ramp-up assembly production lines. It is result of cooperation of 14 partners; software companies, universities and main customer: AIRBUS.<sup>1</sup> 3 years project started on September 2012 with total budget over 11M€. Final product will be managing production in dozens of workshops producing thousands orders per day.

In usual scenario of adaptation of SOA, companies with working environments transform their business logic to service oriented model in order to achieve better manageability, flexibility and in order to save financial resources in long term view. The prospective of this business move is based for example on claim, that up to 90% [2] This is partially case of ARUM. Main ARUM use case is management of production of Airbus A350 airplane, system is based on legacy systems used in production of previous AIRBUS airplanes. Other use case is Iacobucci aircraft interiors production.

Key desired features are scalability, integrated control of production features, multilevel optimization and system stability without single failure points. ARUM additionally provides intelligent tools and framework for integration of other systems, such as multi-agent systems. This integration comes in naturally since MAS and SOA use same information and functionality distribution and they are implemented with respect to the same practices.[3] Scheduling was chosen as best natural model describing manufacturing problem.

Goals of project ARUM are following:

- **Adaptive planning and scheduling.** ARUM system is equipped with multitude of services that in cooperation create system capable of real time adaptive planning according to current state of production line and all resources involved. These services are Scheduler service, using multi-agent paradigm to solve the scheduling problem, Ontology service that organizes knowledge base for planner and scheduler services.
- **Optimization of production.** Optimization of production is provided during planner computation, instead of any plan or schedule planner aims at for suboptimal solution, as optimal as possible in the given computational time. General scheduling is proved to be NP problem[4], thus disabling assurance of optimal solution, leaving us with probably suboptimal solution.
- **Business processes improvements.** ARUM system deployment and usage will provide large amounts of production data. Using data mining methods or graph analysis can yield interesting and useful information that can lead to future improvements and optimizations.

---

<sup>1</sup>These companies are EADS, AIRBUS, CERTICON, Iacobucci, TIE Kinetix, SMRT, ALM, CUAS, P3 group, UNIMAN, IPB, ICCS, UniHa [[arum:site](#)]

Three components implemented are firstly deployment tool, used by administrators to effectively deploy and undeploy components to the ESB, secondly Jade gateway library, allowing Jade platform agents to communicate with ESB services and vice versa and thirdly database service that encapsulates multiple database engines and types and unifies access through common protocol and provides databases as a service.

## 1.1. Airplane manufacture

ARUM main use case is production of Airbus A350. Aircraft production lines are besides shipyards one of the most complicated construction and assembly chains.

### 1.1.1. Airplane construction

[5]An Airbus airplane manufacturing process is based on highly-efficient cooperation of company’s international supply and manufacturing chains. Interconnected network of smaller to bigger production plants passes bi-products of aircraft manufacture to each other leading eventually to assembly of whole aircraft. Especially transportation and passing on assembled sections of airplane proves itself to be challenging.

All Airbus airplane semi-assemblies eventually find their way to the final assembly lines in Toulouse or Hamburg, where they are transported using Airbus Super Transporters. But before that all the pieces from all the manufactures spread across the Europe must be at the right place at the right time. Image 1 shows locations of manufactures needed in Airbus A380 construction process and sub-assemblies transportation pipelines.

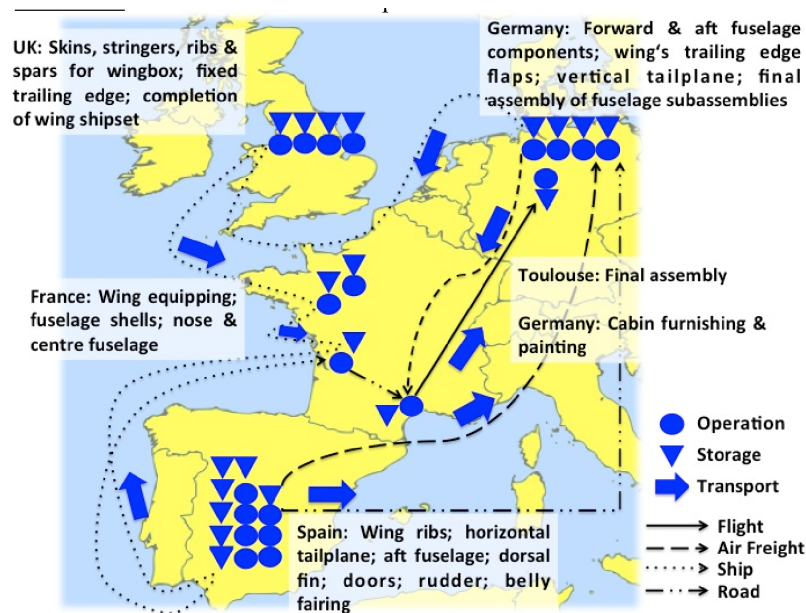


Figure 1. Map of manufacture locations around Europe used for construction of Airbus A380[6]

### 1.1.2. Manufacture planning

Each node of production process can be thought of as a black box that converts input resource components into product, that is input component for a different node. All of

these nodes are organized in a DAG (directed acyclic graph).

One of the main problems that occur in this kind of production process and can be especially troublesome in processes in a scale similar to Airbus is when one of the nodes doesn't deliver its product on expected time. From that point all subsequent nodes can be unable to continue the process. This can happen due to malfunction, resource depletion or human entropy.

We can call the manufacturing process robust, if the whole system continues on working even though some of its components are occasionally out of service. This is one of the scenarios that usage of ARUM system should prevent happening by means of resource rescheduling, bottle-neck detection and other.

Information distribution is crucial for manufacture planning and scheduling. Any planner needs data about current state of manufacture as well as units participating in manufacture must receive their assigned schedules.

### 1.1.3. SOA integration

Main idea behind service oriented architecture is separating managed process into several disjunct subjects, each of those not directly dependent on one another, label those subjects services and allow them to communicate in common way. Each industry process that is transforming for SOA integration must modeled in SOA scene, that means separating the process on small independent subjects.

Question here is what granularity to choose, higher the granularity the higher flexibility but whole process gets more computationally demanding. It is trade off between bias and variance.

### 1.1.4. ARUM

ARUM project faces challenge of integrating whole production process of manufacture of Airbus A350 aircraft, covering multiple factories/assembly lines, managing human resources on each work shift, managing material resources. Allowing human administrators oversee the processes, initiate planning and scheduling computations and manage the results. For long term usage ARUM allows collection and simple analysis of production data on the fly.

In case of ARUM project granularity for the production model was set to smaller scale, allowing it management of each employee and each piece of material figuring in the process. Planners use internal virtual model of manufacture and simulates the process to find best solution. This computation is decentralized and uses multi agent distributed optimization. Computation is capable to update data during scheduling computation. ARUM schedulers will adaptively learn from previous computations and this knowledge will speed up future computations.

## 1.2. ESB

[7] Enterprise service bus (ESB) is a software architecture that provides communication between mutually interacting software applications, it is one of the SOA implementations. It is a tool for integration of heterogeneous applications in enterprise environment.

Concept of ESB took its inspiration in computer hardware, where 'bus' represents channel for communication, managed by arbiter, where any component can be connected and is able to communicate with the rest of the bus, without prior preparation

of the bus for connection of the new component. To mimic this functionality ESB engine, the bus arbiter, has these functions / responsibilities:

### **Message routing**

ESB provides communication channel for all the services to interact with each other. Communication is done using messages and it is ESB responsibility to deliver these messages to designated addresses. Message is data package, containing body with useful data and header carrying meta-data and routing options. This also assumes existence of service directory or similar, interface for the services through which they query addresses of other services on ESB. Message delivery is often implemented using message queues and topics. Queues are by default persistent and ESB guarantees delivery to rightful service, even if it is currently undeployed, not running or it crashed during processing of the message.

### **Service deployment and versioning**

ESB provides tool to add or remove running services from ESB.

### **Event handling, data transformations and other fancy functions**

ESB occasionally comes with additional features, which purpose is to allow user integrate their systems without need of actual coding. This for example is data transformations, implementation of various service and process interfaces, protocol conversions, event management, security and exception handling.

## **1.3. ARUM iESB**

[8]ARUM integration engine is based on JBoss implementation of SOA. JBoss Enterprise SOA Platform (JBossESB) is open-source Java Enterprise Edition based SOA engine. It contains portfolio of business process tools, such as bRULES engine[8], HornetQ[9] messaging software, process monitoring, business process management(BPM)??, directory services with security and it is capable of clustering.

JBossESB is frequently used implementation of ESB and it has large community. It runs on JBoss application server (JBossAS). JBoss is currently developing another SOA solution called SwitchYard, which is intended to eventually replace JBossESB but it is still in it's earlier versions of development. seamless integration

ARUM extends existing JBossESB by various intelligent tools, turning ESB into iESB (intelligent ESB). Purpose of these tools are to provide seamless SOA integration, allow easy data and process management and simplify system scaling. iESB also gets more involved in content and format of data and services deployed on it.

These are some of the core differences between ESB and iESB:

- **Liferay** [10] Liferay portal is a free and open source web portal created in Java. It allows creation of customizable websites and web portals assembled from portlets, small chunks of web content and execution logic. Liferay is bundled with servlet container Apache Tomcat[11]. Liferay extends iESB by graphical user interface to underlying services, it allows interaction, management and administration of ESB for human users through web portal that Liferay provides.

Deployer component, which is first focus of this thesis, is mostly setup on Liferay part of iESB.

- **ARUM message format.** Services on JBossESB communicate using ESB messages, which compose of mainly of header and body, where body can contain unspecified and unrestricted data. Header contains message's meta-data, but only the target address is required. This allows a simple and quick implementation but it has its downsides. Working with an unspecified messages forces separate logic implementation for each communication path.

Messages without set format can't be effectively analyzed and organized in an universal manner. Communication on iESB in ARUM uses an extended version of ESB message called ARUM message. ARUM messages forces user to define additional message meta-data, full routing information, restricts the content to character string only and forces usage of conversation ID.

Additionally ARUM adopts FIPA[1] communication protocols. FIPA is used in multi agent systems (MAS), where it defines standard possible means of communication between agents in the system. This makes development more guided and controlled, reducing risks of dead-locks[12] and loss of messages.

Messages in accordance with FIPA standard must additionally include information about what protocol is used (predefines communication procedure) and message's performative(position and status of agent in the pending communication).

This means services can only use FIPA predefined communication protocols to cooperate with each other. Such restriction proved itself to be well-arranged for large implementations, easier to debug and more accessible to network communication analysis of the ARUM Sniffer.

For the use Jade gateway 3, message header was extended by another routing option; Agent name, to allow agents in MAS embedded into iESB to freely communicate with one another.

- **Deployment tools.** Default implementation of the JBossESB and underlying JBossAS provided little means of interactive deployment. When talking about deployment (and undeployment) of services we mean transferring new piece of program onto JBossAS and setting it up as a new service in the JBossESB.

Services are deployed onto ESB using .esb packages. ESB package contains mainly definitions of message queues, services, compiled service logic and other configuration files. Chapter 2 describes implementation of custom deployment using custom ARUM packages, that contain standard .esb packages bundled with other archives like portlets etc..

Deployment tools allow analysis of these ARUM packages, deployment and undeployment, and control modifications in iESB that are necessary before deployment of new a service.

- **Sniffer.** Sniffer is iESB component that tracks and aggregates all communication on iESB, it is content and FIPA preformative aware and it tracks all the motion on iESB. ARUM system deployment and usage will provide large amounts of production data. Restriction of ARUM messages only to FIPA protocols is useful for the indexation.

Sniffer is accessible through graphic user interface (GUI). Provided through Liferay portal as a portlet.

[2]

### 1.3.1. Other iESB contents

These are other important services and tools available in the ARUM system.

#### **Ontology service**

Stores ontologies used by scheduler and other components operating with plans and schedules. Ontology service also dynamically expands its knowledge from all successfully computed plans and schedules thus learning and helping make future computations faster.

#### **Scheduler**

Scheduler is the core feature of ARUM. Given status data it computes schedules as optimal as possible and gives those schedules to manufactures. It is also capable of predicting and what-if planning, which can discover potential loopholes and bottlenecks and help computations done in future (long term planning and scheduling).

Computation is agent-based in combination with constraint logic programming method for optimization (CLP).[13]

#### **Factory network designer**

GUI tool for creating semantic models of workshops(workers, lines, machines). This data is feeded to ontology service and it is used as an input data for planning and scheduling.

#### **Scenario designer**

GUI tool for creating semantic models of alternative ramp-up scenarios and suitable risk and contingency management strategies[14]. It also feeds final data to ontology service used in planners.

#### **Jade gateway library**

Jade gateway is java library available for services that want to use Jade agent platform. Agents are this way enabled to communicate with iESB services. Implementation is described in chapter 3.

#### **Database service**

Database service provides data storage for all other services. Implementation is described in chapter 4.

## 2. Deployer

Deployer is core component of ARUM iESB platform. It provides extended possibilities of manipulation with other iESB services comparing to JBossESB. ARUMDeployer source codes are available on attached CD.

### 2.1. Analysis

ARUM system creates environment where various services work together to manage work of a production process. Some of these services need human input, in cases like new scheduling request, database update after material stock at the production line has been replenished or when worker gets sick. Services that bring this information into the system have their front-ends to allow human users to interact with them.

There are multiple way of implementing the front-end, for ARUM project web portal and access via HTTP was chosen and implementation of web portal used is Liferay portal[10].

Liferay is an enterprise web portal, that provides web access with user authentication, custom portal for each user, modular architecture that uses "portlets" and other.

Each iESB service that needs human input is implemented with another component to it, and that is Liferay portlet. When both portlet and service are deployed, one on JBossESB and the other on Liferay, using web interface, user that is authorized to use the portlet can now manipulate the service in a way that the portlet was designed.

Problem lies in separate deployment. Both portlets and services must be deployed and undeployed together to prevent errors as well as updated together. Both Liferay and JBoss support deployment that uses file system.

For this reason ARUM iESB needs unified package that stores both portlet and service, and iESB component that can work with such packages. Front-end of this component must be in form of portlet, accessible through web.

This component must be able to:

- Upload the integrated packages and provide information about the package to the administrator.
- Allow the administrator to choose components that need to be deployed, deploy both JBoss and Liferay parts of the iESB component.
- Discover already uploaded components on the server, deploy or undeploy these components.

### 2.2. ARUMPackage

ARUMPackackage is realization of desired custom package.

It is a file compressed using ZIP[zip ] method. ARUM Package can contain multiple iESB components at once. Zip can contain:

- **Multiple tools** Each tool is represented by single folder, folder contains one or more archives deployable on either JBossESB or Liferay.
-



## 2. Deployer

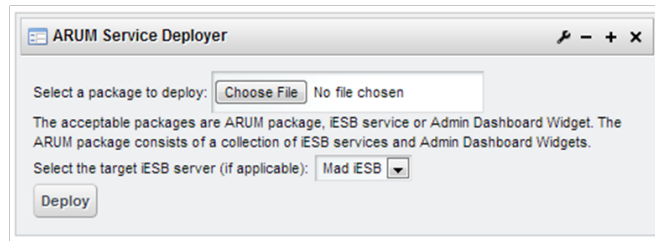


Figure 2. Deployer portlet, Upload UI.

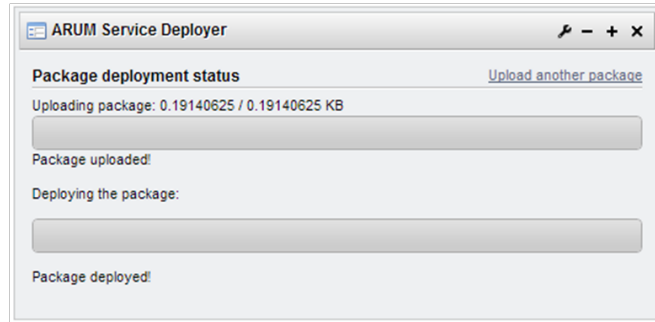


Figure 3. Deployer portlet, Upload UI after successful upload

Figure 4. Deployer portlet interface[15]

### 2.3. Deployment process

Liferay packages are for example .war files, or any archive deployable on apache tomcat [11]. JBossESB packages have extension .esb.

#### 2.3.1. ARUMPackage to XML

First part of the deployment process is upload of ARUM Package to Deployer component on Liferay

### 2.4. DeployerPortlet and ARUMAdminService

Two main parts of the ARUMDeployer are deployed on Liferay portal.

First one is the DeployerPortlet. This component is being developed by other members of Certicon development team. It's purpose is to provide GUI web interface for the system administrator and invoke functions of the other parts of ARUMDeployer. Figure 4 shows appearance of deployer portlet during upload.

Second part of the ARUMDeployer on Liferay is the deployment logic implemented as a part of this thesis.

It is a part of larger ARUMAdminService component and it provides functions of ARUMDeployer as a web service.

These functions are:

- **UploadFile** This procedure receives the ARUMPackage from the DeployerPortlet using HTTP POST[16].

ARUMPackage is uncompressed and contents are processed in a way that is described in section 2.3.1. Procedure returns XML string that represents contents of the package <sup>1</sup>. All the tools and their Liferay and JBossESB components are

<sup>1</sup>Design of the XML can be found on attached CD in configuration/ARUMDeployer section.

## 2. Deployer

stored in archives on the Liferay machine's file system. Deployer configuration sets this location using `DeployableObjectsStorageFolder` field in configuration.

Packages stored in deployable objects folder are ready to be deployed by calling `Deploy` procedure, each archive has its own unique ID that identifies it. Using this ID, other functions like `Deploy`, `Undeploy`, `DeleteArumArchive` or `GetArchiveInfo` can use.

- **Deploy.** When `deploy` is called, procedure finds the desired archive and depending on whether it is archive designed for Liferay's tomcat or JBossESB it calls subsequent function and tries to deploy the archive.

In case of Liferay, archive is simply moved to Liferay's tomcat deploy folder defined in the configuration by `LiferayDeployFolder` value. Tomcat starts the deployment automatically.

JBoss deployment is more difficult. Part of `ARUMDeployer` is JBoss service that is capable of receiving files from `ARUMAdminService`. Using this service, esb archive is transported onto the JBoss server and deployed. Section 2.5 describes the service in more detail.

- **Undeploy** Undeploy procedure is inverse of `deploy` procedure. This means it deletes the desired archive from tomcat deploy folder in Liferay case and through service on JBoss it deletes it from remote deploy folder in JBoss case. This however doesn't delete the archive from folder with prepared archives, after archive was undeployed, it can be immediately redeployed without need to upload it first.
- **DeleteArumArchive** Deletes one of the archives from `DeployableObjectsStorage` folder. This will remove the ability to deploy the archive as well as undeploy the archive, since there is no reference to what are the meta-data of the archive needed to undeploy it. This means undeploy procedure must be executed before removing the archive in order to completely getting rid of the component.
- **GetArchiveInfo** Returns information about the prepared archive in XML format.
- **GetArchives** Returns list of prepared archives, that are uploaded on the server.

Procedures `GetArchiveInfo` and `GetArchives` were based on architecture implemented for this thesis however they were implemented by the rest of the Certicon team recently.

### Configuration

Configuration is stored in web service's resource folder in file `config.cfg`. These configurations include:

- **DeployableObjectsStorageFolder:** Points to location on file system, where uploaded packages are to be stored.
- **LiferayDeployFolder:** Points to Liferay's tomcat deploy folder, archives moved to this location get deployed onto tomcat.
- **JbossFileTransferPacketSize** Size of packets that are sent to JBoss when transferring the archives.
- **JbossSyncMessageDeliveryTimeout:** Timeout of synchronous part of file transfer from Liferay to JBoss.
- **JbossDeployerServiceName:** Name of the `JBossDeployerService`.
- **JbossDeployerServiceCategory** Category of `JBossDeployerService`.

## 2.5. Deployer service

Since most of the logic is done in ARUMAdminService side of ARUMDeployer. JBoss side, JBossDeployerService, is a ESB service that sole purpose is to either receive a file and move it to different folder or to delete a file.

Archive is deployed by receiving it through iESB and moving it into the JBossESB deploy folder. This folder is defined in configuration.

When archive needs to be undeployed, ARUMAdminService sends ARUMMessage, that simply states what is the name of the archive that needs to be undeployed. DeployerService simply deletes the file.

### 2.5.1. Simple message based file transfer

Using very simple protocol, JBossDeployerService can receive esb archive from ARUMAdminService on Liferay and deploy it.

Protocol is as follows:

1. ARUMAdminService initialized the protocol with ARUMMessage, that is sent in synchronous transfer. This message carries: size of the payload in bytes, name of the archive<sup>2</sup> and ID of transfer.

The ID is important since multiple file transfers can be active at one time, and the service is capable to take care of all of them.

DeployerService remembers the ID, opens new file with given name, returns success message and starts waiting on next message.

2. Further messages are sent asynchronously to speedup the transfer. When next message with matching ID arrives, it is either next expected message or one of the messages got stuck<sup>3</sup>, in that case each file transfer stores list of messages that must be put together to form the archive that is being sent.

By increasing packet ID of the next expected message, file transfer one by one accepts ESB messages and byte content, encoded into message's string content, gets written into the file.

No response message is sent.

3. Last received message is also sent using synchronous transfer. DeployerService checks, if the expected size of the file matches the actual transferred size. Size not matching would suggest corrupted file transfer.

DeployerService closes the file and moves it into the deploy folder, finishing the remote deployment process.

Returns success message to the ARUMAdminService.

When the archive was transferred, it is moved to the deploy folder of the JBossESB, where JBoss takes care of the archive and deploys whatever is contained in the archive.

## 2.6. Under development

### 2.6.1. Portlet

Portlet is still under development, in the future it should allow the user full interactive interface, providing list of available archives and deployed archives, browse the properties etc.

---

<sup>2</sup>Name that should be used to store the archive on JBoss side. Using this name the archive can be undeployed in future.

<sup>3</sup>This scenario shouldn't happen on JBossESB, however DeployerService still covers this possibility

## 2. Deployer

Important feature that is to be implemented is cooperation with ARUM Database Service (ADS). Administrator that uses the deployer portlet must be able to determine what authorizations on the data each tool or archive requires and what are the deployment possibilities. It should be able to cooperate with Database portlet<sup>4</sup> and decide what to deploy.

This also influences ARUMAdminService, which will need to implement administrator interface of ADS, creation, edition or deletion of ADS configuration or databases inside the ADS. Chapter 4 is dedicated to ADS.

### 2.6.2. Extension of functionality

Further capabilities may include administration of ARUM Database Service(ADS), package analysis, actual security provided by SecurityService that is under development and others.

The deployer service doesn't have direct access to both JBoss and Liferay output consoles and so it doesn't have the access to error messages in case the deployment was a failiure. Deployer should be upgraded so it provides the administrator with the error messages.

---

<sup>4</sup>Portlet that will allow administration of the ADS

## 3. Jade gateway

This chapter describes the implementation of a library that allows integration of Jade platform into ARUM iESB. The whole name of the library is ARUMJadeGateway.

This project is stored on the CD under projects/ARUMJadeGateway.

### 3.1. Analysis

Multi agent approach was chosen for ARUM schedulers and planners as the most perspective method with the highest development complexity / effectiveness ratio. MAS allow for the use of the divide and conquer method for solving difficult tasks and the implementation of a single logic part is much easier and doable.

The second part of the problem was how to integrate schedulers into iESB system, where they can communicate with other parts of the production, be queried and generally be useful. Since ESB and Jade are incompatible systems, middleware has to be created in order to integrate services and agents.

#### 3.1.1. MAS vs ESB

Even though MAS and ESB are different and independent systems their internal functionality is somewhat similar.

Both systems are organized into independent computational bundles called services or, in case of MAS, agents. Both agents and services are computationally independent, having their own program threads. Both of them interact with the rest of the platform using messages, which can be sent to other fellow services or agents. They both receive messages and react to them.

In both MAS and ESB, time synchronization is not assured since each agent runs on its own clock, message delivery time varies, however, messages come in the same order as they were sent.

The integration of MAS agents into ESB includes:

- Message content, protocol and performative, where performative describes state of the conversation in given FIPA protocol [1]. conversion. Conversion must be symmetric in both ESB to MAS and MAS to ESB conversion.
- Message routing. Messages must arrive to designated receiver. If one of the systems creates and sends a response, it must arrive to the original sender.
- Synchronous/Asynchronous message delivery mirroring. To finish up seamless message delivery, both sides must be able to use all message delivery options.

#### 3.1.2. Jade

Jade (Java Agent DEvelopment Framework) is a software Framework implemented in Java. It provides middle-ware for FIPA specification compliant agents and provides a variety of GUI tools that simplify debugging and development. It is multiplatform, distributable and remotely controllable via GUI interface.[17]

### 3. Jade gateway

Agents implemented in Java communicate through platform using FIPA-ACL language

The agent platform can be distributed across machines (which do not even need to share the same OS) and the configuration can be controlled via a remote GUI. The configuration can even be changed at run-time by moving agents from one machine to another one, whenever required.

#### 3.1.3. Jade vs. iESB

Jade uses ACL messages. ACL messages are an implementation of the FIPA-ACL protocol. Each message contains most importantly string content, the receiver address, the sender address, the FIPA protocol and the FIPA performative. Protocols and performatives are discussed in 3.2. Since iESB implements the FIPA message format in ARUM messages, the conversion method is obvious. The difference in messages is in the receiver address.

The receiver is on a Jade platform, is always an agent and is defined by an agent name and platform address pair.

The Jade platform consists of at least one agent container, which is then called the Main container. Additional containers can be created on different machines and they can be connected to the main container. This set of containers then shares a platform and all agents living inside each of those containers can communicate with each other as if they were actually in the same container. This fact is used in the implementation of the Jade gateway.

Additionally Jade allows communication of two agents from different platforms as long as one of the agents also knows the network location of the other platform. We could use this feature to enable the communication of multiple MAS systems, however agents wouldn't be able to communicate with services on iESB, or more precisely, services wouldn't be able to contact multi agent systems.

Jade's equivalent of service discovery is a Directory Facilitator (DF) agent. DF manages and provides a directory that contains all agents available on platform.

iESB uses two ways of communication, synchronous and asynchronous, whereas Jade only uses asynchronous transfer. Therefore the Jade gateway must implement a function that converts synchronous communication into asynchronous.

One of the minor differences between ESB and Jade is assured delivery when sending messages through JBossESB. iESB tries to repeatedly send a given message until the receiving service successfully accepts it. Messages for services that are not currently available are stored. This could potentially lead to message duplication in cases of synchronous transfer on iESB side and asynchronous transfer on Jade side.

## 3.2. Implementation

This section describes the implementation of the Jade gateway (JG), a bridge between ESB and Jade messaging spaces, and how to use it.

### 3.2.1. Architecture

Two main components of JadeGateway library are a main object managing message routing from iESB called the `JadeGateway` object and an agent counterpart deployed on the Jade platform, Jade gateway agent (`GatewayAgent`).

### 3. Jade gateway

The Jade gateway communicates with agents on MAS by sharing the same Jade container or the same Jade platform and deploying a Jade gateway agent (GatewayAgent). The functionality of this agent will be described later in detail. The main purpose of GatewayAgent is to relay messages from JADEGateway to iESB.

However, JADEGateway is only a library that can be used by other services to make MAS integration into iESB possible. This means the user of this library must create the iESB service that passes traffic from iESB to JADEGateway. This service will be referred to as the `EntryPoint` service. To help the user with the implementation of a MAS compatible iESB service, the JADEGateway library provides classes that implement the functionality of configuring, initializing and operating with JADEGateway, reducing user input to only providing configuration.

One of the problems that JG has to deal with is JVM incompatibility between the JBoss application server and some of the schedulers and planners.

JBossESB used in the ARUM project is compatible only with an older JBoss application server, which is JBossAS 6. This application server uses Java version 1.6, comparing to newer version JBossAS 7, which uses Java version 1.7. Thus everything running on iESB must be Java 1.6 compatible.

However some of the schedulers, for example the latest version of Choco solver[18] library for ARUM Scheduler is written in Java 1.7 and obviously fails to load on JBossAS 6.

The problem is how to connect the Java 1.7 Scheduler to the Java 1.6 iESB. The solution created in Java gateway is based on a feature of the Jade platform.

Jade is written in Java 1.6, so it can be executed on both Java 1.6 and 1.7. The proposed architecture is based on multiple JVM's.

The ARUM service, which uses JG, starts on JBoss and by initializing JG it creates a Main container with a GatewayAgent. A secondary container can then be created in a different JVM, in Java 1.7 if needed, and it connects to the Main container. Now agents on both containers can communicate with each other seamlessly. The agents that want to contact iESB do it through GatewayAgent and traffic coming from iESB gets into the Jade platform through GatewayAgent as well.

There are multiple ways to setup a secondary Jade container in JADEGateway architecture:

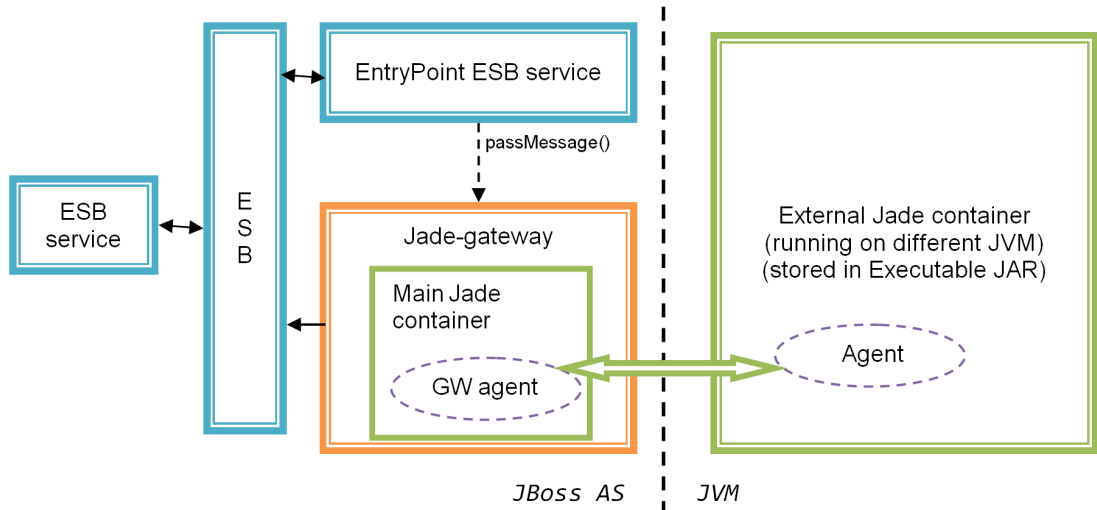
- The default setup that JADEGateway was firstly designed for is supplying a JAR file to the JADEGateway. This JAR file should contain the code to start up the external container. After configuring which JVM to use and initializing JADEGateway, the given JAR is executed and provided information about where to connect to the main container. When the `EntryPoint` iESB service is undeployed or stopped, it also kills the external Jade container by killing the created JVM.
- The gateway can be set up to skip external container startup and run only the main container. All other containers must be run manually by the user or a system administrator. This way JADEGateway cannot give an address to the external container and theoretically the external container doesn't know where to connect to. In practice, the service side with the main container is preconfigured and the external container is aware of this configuration and connects to it.
- Another possibility is a hybrid of the first two possibilities. If JADEGateway starts a new process that instead of creating new container contacts other machines on the network and announces that the iESB services have been started, these machines can react to this, start external containers and connect to the machine

### 3. Jade gateway

where iESB with a JADEGateway enabled service operates. This way we have created a distributed and possibly even heterogenous Jade platform.

- As the last possible solution all agents can be located on the main container, but this is the least used option because it includes extending of the JADEGateway class.

The architecture is shown in figure 5. Blue color represents iESB JBoss space, orange represents the JADEGateway library and green represents the Jade platform. A dashed line separates two JVM's, black arrows show iESB communication and a green arrow shows agent communication on the Jade platform.



**Figure 5.** Jade gateway architecture using multiple JVM's.

#### 3.2.2. Gateway

JadeGateway represents the main class of the JADEGateway library. It holds the Jade container and controls the Jade platform and the gateway agent.

Code sample 3.1 shows most important functions of JadeGateway:

- The PassMessage method accepts a message from iESB. It routes it according to routing options described in 3.2.2.
- RegisterNonMASReceiverCallback defines the default routing option for unrecognized messages as an external object implementing interface, more about routing in 3.2.2.
- RegisterNonMasReceiverDefaultAgent defines the default routing to be sent to a given agent on the platform.
- InitJadeContainer initializes and starts the main container and the JADEGateway functionality. This also starts the external JAR file if specified.
- The StopJadeContainer and Destroy methods are similar procedures, which take care of shutting down the gateway in case of just stopping the gateway or in case of iESB undeployment.
- SetJarPath sets a path to the JAR file that will be started after the gateway initializes. Its typical content is the code for starting up a secondary Jade container. For the easy implementation of a secondary container, the user can extend the



### 3. Jade gateway

provided `ESBAwareSideContainer` class, which takes care of parsing the input, connecting to the main container and creating the external container for the user.

Code 3.1 Part of `JadeGateway` class interface

```
public interface JadeGateway {
    ...
    public ARUMMessage passMessage(ARUMMessage message);
    public void registerNonMASReceiverCallback
        (NonMASReceiver register);
    public void registerNonMasReceiverDefaultAgent
        (String agentName);
    public void initJadeContainer();
    public void stopJadeContainer();
    public void destroy() throws ActionLifecycleException;
    public void setJarPath(String pathToJAR);
    ...
}
```

The `JadeGateway` can be easily created and initialized using the `AgentAwareESBService` class described in 3.2.6. By extending the `AgentAwareESBService` class the user receives a functional iESB service and an initialized `JADEGateway`.

#### Routing

The gateway implements these routing options for communication. They can be changed before the initialization call of the `JadeGateway` class.

1. **Recognized CommunicationID.** Each message on both iESB and Jade platform contains a string property in the header called either `CommunicationID` for iESB or `ConversationId` for Jade. `JadeGateway` uses the `CommunicationTable` class to keep track of messages going through the gateway. `CommunicationID` stays the same for all the messages in a single back and forth conversation in all FIPA protocols. This means that if MAS sent out a message with a given `ConversationId` and later a message arrives with the same `ConversationID`, it must be a response to the previous message and it is supposed to be routed to the sender of the original message. First off, `JadeGateway` asks the `CommunicationTable` whether a message with the given `CommunicationId` is expected (was relayed before), and if it is, it gets routed accordingly. Otherwise, the routing proceeds with the second rule.
2. **New conversation.** If the iESB service tries to start a new conversation, it must specify what agent to contact. This is done through one of the fields of `ARUMMessage` that stores the name of the agent; `agentName`. If this field is not null then the gateway creates a new conversation entry in `CommunicationTable` and routes the message to the specified agent.
3. **Default routing.** If the message is neither a response to a message sent from the gateway nor a message addressed to an agent that creates a new conversation, then default routing takes place. Default routing must be configured before `JadeGateway` is initialized. It is configured to either route to the default agent or to be routed to a configured object.

### 3. Jade gateway

The `registerNonMasReceiverDefaultAgent(String defaultAgentName)` function sets the default routing to a default agent. That means unrecognized messages get routed to the agent specified.

The `registerNonMASReceiverCallback(NonMASReceiver receiver)` function sets the default routing to the given object implementing `NonMASReceiver` interface.

This interface defines only a single method that receives `ARUMMessage`.

If default routing was not specified, the gateway throws an exception.

#### Configuration of the gateway

This section enumerates all the information that should be configured in `JadeGateway` before initializing it.

- The default routing using either a default agent or providing a `NonMASReceiver` object, more information in 3.2.2.
- Executable JAR or disabling startup of secondary container by using `setJarPath` or `enableSideContainer` methods.
- Defining a TCP/IP port that the main container will be listening on. By default the port is 11010. Port is set using the `setPort` method.
- Defining a service name and category. `ARUMService` that nests the gateway.

#### Communication table

The `CommunicationID` stays the same for all messages in a single back and forth conversation in all FIPA protocols. Communication table stores `CommunicationIDs` of the active conversations in form of `ConversationBinding`. Message that arrives with the same `ConversationID` as one of the active `ConversationBindings`, it must be in the same conversation and it is routed to one of the saved endpoints.

Codes in 3.2 and 3.3 show parts of classes involved. `ConversationBinding` represents a single communication across iESB and Jade. `ThreadStopQ` is a class that is used in the SynchronousToAsynchronous protocol, more information about this protocol can be found in 3.2.5.

`createNewId` is called when a new communication is initiated.

There are two way to destroy the `ConversationBinding`, that is recognizing message as a last message in the FIPA protocol. According to FIPA protocol, the same communicationID cannot be used for different conversations. This means that once the FIPA protocol was finalized, the gateway doesn't have to remember the `conversationBinding` and it can close the communication channel by removing the instance from the `cbTable`. The second way involves timeout. If a binding is inactive long enough, it is removed. Information about inactivity is stored in the `time` field, which represents a timestamp in milliseconds when the binding was used the last time.

Functions `expects` are used during first rule of routing 3.2.2 when the gateway decides wether the message belongs to one of the existing conversations. And subsequently if the gateway does expect the message, using the `getConversationBinding` gateway fetches the information about the endpoints figuring in the conversation and routes the message accordingly.

#### Code 3.2 CommunicationTable

```
class CommunicationTable {...
    private ConcurrentHashMap<String, ConversationBinding>
        cbTable;
```

```

    private ThreadStopQueue threadStopQ;
    //These messages create new conversation binding in the
    cbTable.
    public String createNewId(ACLMessage initiationMessage)
        {...}
    public void createNewId(ARUMMessage initiationMessage)
        {...}
    // First routing rule for both iESB or Jade message
    public boolean expects(ARUMMessage message) {...}
    public boolean expects(ACLMessage message) {...}

private ConversationBinding
    getConversationBinding(ARUMMessage message){...}
private ConversationBinding
    getConversationBinding(ACLMessage message){...}
...}

```

Code 3.3 ConversationBinding

```

class ConversationBinding{...
//CommunicationID
    private String id;
//Remote iESB service
    private ARUMServiceEndpoint service;
//Name of local agent that is participating in the
    conversation
    private String agentName;
//Remote agent, can be null
    private String remoteAgentName;
//Last time this object was queried
    private long time;
...}

```

The CommunicationTable also contains code for overseeing synchronous to asynchronous conversion and ThreadStopQ instance to block threads waiting for synchronous response.

### 3.2.3. Gateway agent

The Gateway agent is an implementation of a Jade agent and it is a counterpart of the JadeGateway class. It is nested inside the main container and it allows agents to communicate with iESB. All messages that are sent from MAS to iESB are first patched through GatewayAgent, as well as from the point of view of other agents, messages that come from iESB look like they are sent from GatewayAgent.

iESB to MAS uses one of the functions implemented in Jade agent called ObjectToAgent(O2A). This function allows the gateway to pass objects to an otherwise unaccessible GatewayAgent instance<sup>1</sup>. ARUMMessages are passed to the GatewayAgent using this O2A function.

<sup>1</sup>Jade doesn't provide direct handles to agents deployed inside the platform. It only provides an object that serves as a remote controller.

### 3. Jade gateway

ACL messages<sup>2</sup> are received by the AgentToObject(A2O) function of GatewayAgent. Communication initiated by agents is a little more difficult than in case of O2A. Since ACL messages cannot hold iESB routing information, the agent must announce future Jade to iESB communication prior to actually sending the message to iESB. This is done using the receiveCommunicationId protocol, which is described in section 3.2.4.

Extract from the GatewayAgent interface is in listing 3.4.

`processO2A(ARUMMessage esbMessage)` processes messages that came from iESB and routes them to the appropriate agents. Routing information is fetched from `CommunicationTable`, where it was stored by `JadeGateway` while processing the very same message or it was stored there previously.

`processA20(ACLMessage aclMessage)` processes messages that came from the other agents on the platform. The message is either sent out to iESB from the agent directly, or passed to a thread waiting for a synchronous reply (details in 3.2.5), or it is the conversation initialization message from `getCommunicationId` protocol (details in 3.2.4)

Code 3.4 Part of JadeGatewayAgent class interface

```
public class JadeGatewayAgent extends Agent {
    private CommunicationTable commTable;
    ...
    public void processO2A(ARUMMessage esbMessage) {...}
    public void processA20(ACLMessage aclMessage) {...}
    ...
}
```

#### 3.2.4. Communication protocols

This section describes possible communication pathways and protocols used in JADE-Gateway.

##### Conversion

ACLMessages(in Jade platform) and ARUMMessages(in iESB) are very similar since they are both inspired/implemented according to FIPA/ACL standard. When serving as a gateway, the discussed library must convert one type to another.

JadeGateway implements a symmetric conversion both ways. The Implementation is inside the `Transformation` class in the methods

`ACLMessage ARUMMessageForAgent(ARUMMessage message, String replyTo)` and `ARUMMessage AgentToARUMMessage(ACLMessage message)`.

Performative id defined in a Jade ACL message is transformed from integer code to ARUMMessage's performative enumeration. This conversion is defined in the `PerformativeConversions` class.

The `PerformativeConversions` class can also recognize whether a message is the last message in the protocol. This information is useful in the `CommunicationTable` where it can justify the deletion of a registered conversationID.

Both messages use only a string as content which makes content conversion easy. All other contents of message header match in type and conversion is straightforward.

---

<sup>2</sup>Messages used in Jade platform.

### Receive communicationID protocol

In order to contact iESB, the message agent must compose a message and have knowledge of the name of the service. Messages cannot be sent directly to the iESB service because the Jade platform will not recognize the address of the iESB message. The Gateway agent must be used as a proxy. This means the receiver of the message is the GatewayAgent but no other fields in the message are expendable. If one of the other fields was modified to carry an iESB address, it could tamper with some protocols used over Jade/iESB.

For those reasons, the Receive CommunicationID(RCID) protocol was proposed.

The agent first announces its wish to communicate with the iESB service. GatewayAgent registers this request, creates a unique communicationID and sends it to the agent. The agent can now use this communicationID in messages and if these messages are sent to GatewayAgent, it recognizes them and routes them accordingly to the destination agreed upon in the RCID protocol.

Figure 6 shows RCID protocol processing.

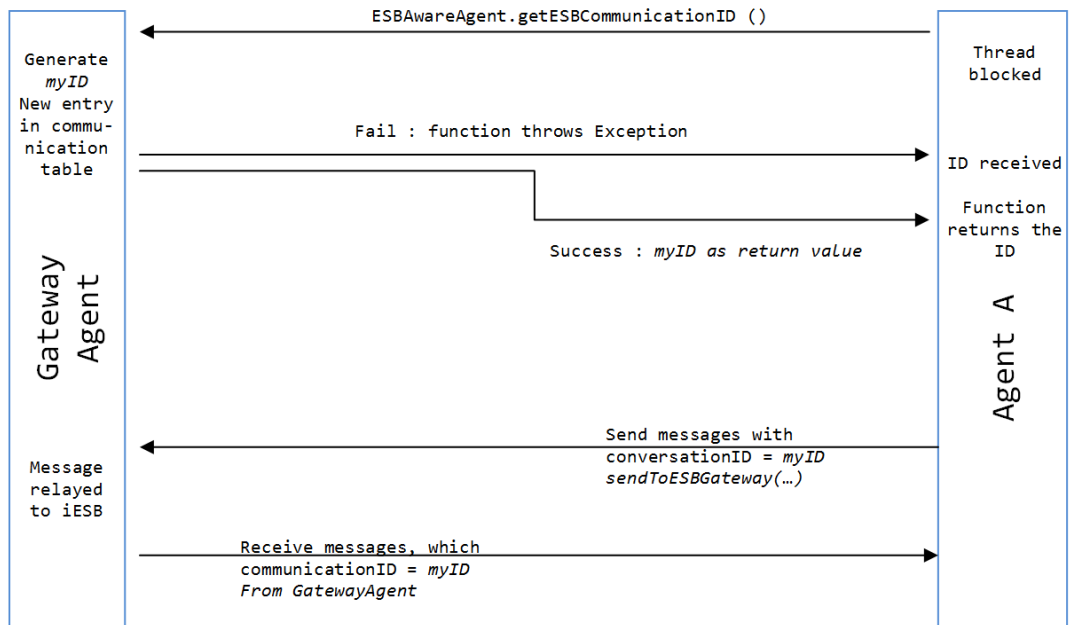


Figure 6. Receive communicationID protocol execution.

### 3.2.5. Synchronous to Asynchronous conversion

The Jade gateway supports conversion between synchronous and asynchronous communication.

When a message arrives to the gateway and the system recognized that message as a synchronous message<sup>3</sup>, instead of passing the message and returning the execution thread, the gateway locks the thread and waits for the response message. As soon as a return message arrives from MAS, the gateway pairs it according to `conversationID`, wakes up the locked thread, passes the received message and allows the thread to

<sup>3</sup>Means of such recognition are still in development, since the message definition is managed by other developers

### 3. Jade gateway

return to JBoss core. From the point of view of JBossESB, the communication was synchronous. The process is illustrated in the image 7.

The conversion uses classes shown in 3.5.

The function `isSynchronous` tries to determine whether the message was in a synchronous way. This isn't as simple as it might look. JbossESB doesn't support any way of determining whether the message was sent synchronously or asynchronously from the receiver's perspective. ESB services are defined as one-way or request-response [8]. Request-response type of service allows synchronous message transfer, however, it doesn't force it, neither can it determine if the message actually is synchronous. This feature had to be implemented in ARUM iESB.

Two main methods for synchronousToAsynchronous conversion are `waitForResponse` used by the iESB thread to wait for response message from the agent and `returnMessage` used by the agent to return the message.

`ThreadStopQueue(TSQ)` uses thread lock to freeze the threads. The class `ThreadStop` is created when iESB synchronous message comes in, the message is sent to Jade and simultaneously a `ThreadStop` instance is stored in the queue and the iESB thread waits inside.

When `returnMessage` is called, `TSQ` stores a response message inside the respective `ThreadStop` instance and releases the thread. Then the thread collects the response message and returns. In case the response doesn't come back before time-out, null is returned and the situation is recognized as a message delivery time-out.

Code 3.5 Classes used in Sync. to Async. conversion

```
public class CommunicationTable {...
    private ThreadStopQueue threadStopQ;
    private boolean asynchronousOnly = false;
    public boolean isSynchronous(ARUMMessage message);
    public ARUMMessage waitForResponse
        (ARUMMessage receivedMessage, AgentController
         jadeGatewayAgentController){...}
...}

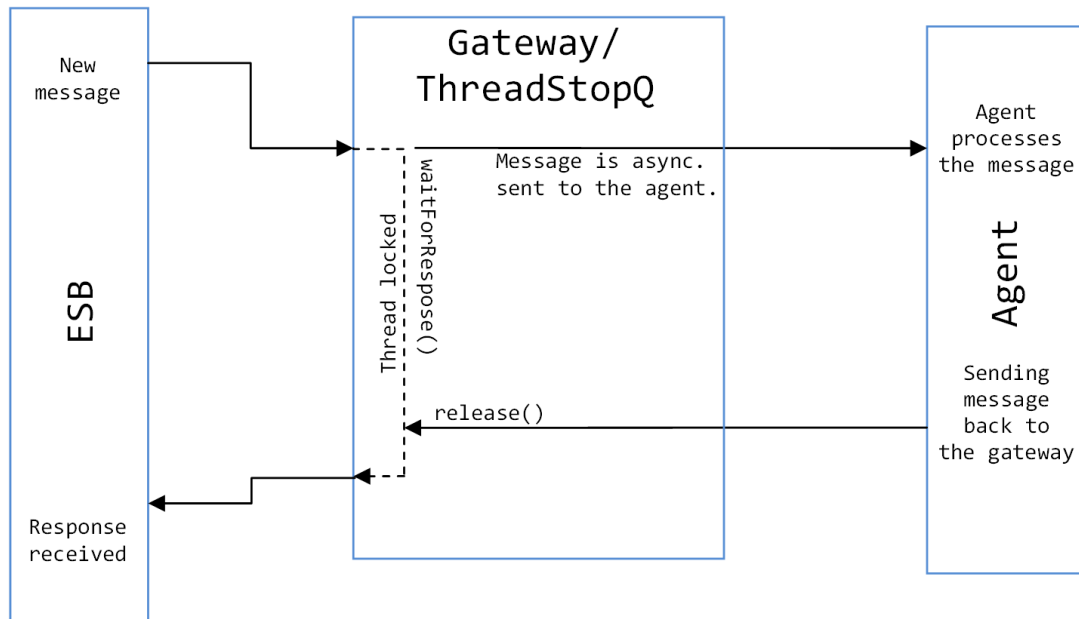
public class ThreadStopQueue {...
    private ConcurrentHashMap<String, ARUMMessage>
        prerecievedMessages;
    private ConcurrentHashMap<String, ThreadStop>
        queuedThreads;
//MAS returns message to iESB thread
    synchronized void returnMessage(ARUMMessage
        message){...}
    synchronized boolean isWaitingFor(ACLMessage
        message){...}
    ARUMMessage waitForResponse
        (ARUMMessage receivedMessage, AgentController
         jadeGatewayAgentController){...}
...}

class ThreadStop {...
    private ARUMMessage message;
    private ARUMMessage response;
    private long timeout;
    private boolean gotResponse = false;
```

```

synchronized boolean waitForResponse(){...}
synchronized boolean release(ARUMMessage response){...}
...}

```



**Figure 7.** Process of converting synchronous communication into asynchronous.

Synchronous to asynchronous communication can be enabled or disabled by calling the `JadeGateway` function:

```
setAsynchronousCommunicationOnly(boolean).
```

Another restriction on synchronous communication on `iESB` lies in the requirement for previous routing information contained in the original message, if this information is not provided (in the form of an instance of the underlying `ESB` message that carried the original `ARUMMessage`), then message sending fails. This problem is easily covered since the thread that carries the original message was blocked until the response came through the asynchronous transfer in the Jade platform, thus it can supply a reference to this message and the whole process continues without complications and the message returned in the synchronous call is passed to the original sender.

### 3.2.6. User API

These classes are bundled in the `JADEGateway` library and they were created to allow as easy as possible development, they represent 3 parts of the `JADEGateway` architecture: Agent-aware service for the front face of the `iESB`-enabled MAS system, `ESB`-aware agent for the agent capable of communication with `iESB`, and an external container class that implements all the logic needed to start a Jade container and connect to an existing main container given commandline arguments.

#### Agent-aware service

By extending this class, the user can easily create an `iESB` service that represents an `EntryPoint` service in 5. The class is capable of managing `iESB` traffic. The user only

### 3. Jade gateway

needs to override the selected methods. Most of the methods are shown in 3.6. Methods whose name start with the prefix `configure` represent procedures that configure the gateway. These procedures can only be called before the gateway is initialized.

The function `configureGateway` is called right before the gateway gets initialized, by overriding this function the user can call various config- methods and thus configure the gateway. Initialization is called automatically by the iESB engine. If the user decides to set up default routing to callback using the `configureDefaultRoutingCallback` function, he might also want to override the

`ARUMMessage receiveNonMASMessage(ARUMMessage message)` procedure, because this is the procedure that gets called once all other rules for routing fail and an `ARUMMessage` is returned back. The function is supposed to response with a reply message.

The user is required to execute the `configureSetService` function. The gateway must be aware of the address of the local service in order to receive messages.

If the user calls the `configureStartExternalJar` function, the given JAR will be started in a new JVM. If the user doesn't call this function, no other JVM will be started.

Code 3.6 AgentAwareService interface

```
public abstract class AgentAwareESBService implements
    NonMASReceiver{
    ...
    private static JadeGateway jadeGateway;
    public ARUMMessage receiveNonMASMessage(ARUMMessage
        message) ;
    public abstract void configureGateway(JadeGateway jg);
    public ARUMMessage receive(ARUMMessage m)
    ...
    //Sets up default routing to receiveNonMASMessage function
    protected final void configureDefaultRoutingCallback()
    //Sets up default routing to a default agent
    protected final void
        configureDefaultRoutingAgent(String defaultAgentName)
    //Sets the name of the service that owns this gateway
    protected final void configureSetService(String
        serviceCategory, String serviceName)
    //Sets the port that the main container will use for
    listening
    protected final void configureSetStartParameters(int
        port)
    //Chooses what JAR to use as a secondary container
    protected final void configureStartExternalJar(String
        externalJar)
    //Restricts the communication to asynchronous only
    protected final void
        configureAsynchronousCommunicationOnly(boolean
        asynchronousOnly)
    ...
}
```



**ESB-aware agent**

ESBAwareAgent is a class that implements Jade to iESB communication from the agent side. Jade agents can either be inherited from this class or they can use static functions to communicate with ESB. ESBAwareAgent implements the receiveCommunicationID protocol described in 3.2.4.

By using the functions in 3.7 the agent communicates with iESB.

Communication starts with the `getESBCommunicationID` function, using which agent receives communicationID. During the call of `getESBCommunicationID`, agent sends message to the `GatewayAgent` and announces future communication with provided iESB service. As a reply, `GatewayAgent` sends `communicationID` string, that will be recognized by the gateway and messages carrying that `communicationID` will be routed to iESB.

After that the agent composes a message for iESB, sets the conversationID of the ACL message to the received ID and sends it out using the `sendToESBGateway` function, which sends the message to the `gatewayAgent`. After message arrives to the `gatewayAgent`, agent recognized the `communicationID` and routes message to iESB, to the iESB service provided during `getESBCommunicationID` call. ConversationTable that now contains the `communicationID` and a pair agent-iESB service is shared between the `gatewayAgent` and the `JadeGateway` class, so messages that come back from iESB as a reply with same `communicationID` will be routed back to the agent.

The user can also use the `sendMessageToEsb` function, which encompasses both of the functions mentioned before.

As for the function arguments: `timeoutMilliseconds` sets how long in milliseconds the function should wait for the `GatewayAgent` to reply. Service category, service name and agent name define the remote address in iESB<sup>4</sup>. `ACLMessage m` is the message to be delivered.

The functions that return string return the `communicationID`.

## Code 3.7 ESBAware agent interface sample

```
public abstract class AgentAwareESBService implements
    NonMASReceiver{
    ...
    getESBCommunicationID
        (String serviceCategory, String serviceName,
         String agentName, long timeoutMilliseconds);
    public static String getESBCommunicationID
        (Agent agent, String serviceCategory, String
         serviceName, String agentName, long
         timeoutMilliseconds);
    public String sendMessageToEsb
        (ACLMessage m, String serviceCategory, String
         serviceName, long timeoutMilliseconds);
    public void sendToESBGateway(ACLMessage message);
    public static void sendToESBGateway(Agent agent,
        ACLMessage message);
    ...
}
```

<sup>4</sup>Agent name is set to null when communicating with the service only.

#### ESBAwareSideContainer

This class implements the creation of an external Jade container and connecting it to the main container. The code sample 3.8 shows the only available static function that accepts commandline parameters as an argument and creates a Jade container that is connected to the main container running on iESB.

Command line parameters work as long as the program is started by JADEGateway and information about network address of the main container is provided.

If that is not the case, parameters must be provided manually. Those two arguments are:

- `-address <address>`: where `<address>` defines the IP address of the main container. In this case it is always “localhost” since Jade-gateway starts a new program locally.
- `-port <port>`: where `<port>` is a number between 1 and 65536 and represents the port number where the main container listens. This number comes from the configuration inside Entry-point service configuration. It is set using the function: `AgentAwareESBService.configureSetStartParameters(portNumber)`; Default value = 10110.

For example startup with manual arguments could look like this: `java.exe -jar MyJar.jar -address 127.0.0.1 -port 12345`

This would start a container that expects the main container to be running at the local machine and listening on port 12345.

After the container is created, the user may work with it in the usual manner.

Code 3.8 ESBAwareSideContainer interface

```
public class ESBAwareSideContainer {
public static AgentContainer GetESBAwareContainer(String []
    args) throws ControllerException {...}
}
```

#### 3.2.7. Interfaces and classes

This section contains a simple explanation of all the important classes that were not previously described in detail.

##### NonMasReceiver

Interface used in JADEGateway routing. A class implementing this interface can receive messages unrecognized in the routing process.

##### ExternalAgentContainer

Class that manages a reference to the JVM process that is supposedly running an external Jade container.

##### Package examples

In `cz.certicon.arum.jade.jadeGateway.example` package, there are examples and tests of agents, services and containers that use some functionality of the JADEGateway library.

### 3.2.8. Future upgrades

FIPA agent integration described in [19] shows integration between a SOA-type architecture and a Jade platform on an even higher degree. This paper describes an architecture where agents living in MAS are discoverable by web services, thus from the point of view of web services, agents are just another web services.

This creates a much more seamless integration between WebServices and Jade and makes extending the system easier.

A future upgrade of the Jade gateway could be allowing iESB services to discover agents existing on iESB services containing MAS systems and vice versa. This wouldn't extend the capability of communication but it would ease the configuration and extension.

The question that arises is whether this is desirable. In MAS not all agents are ready and they are not necessarily expecting traffic from the outside of MAS. Some of the more important agents that are usually the entry points to the system are designed for it, however, the continued operation of the whole system could be jeopardized if unwanted messages were to be processed somewhere deep in the distributed algorithm.

This leads to the conclusion that only white-listed agents should be available for the iESB services to contact. This idea leads to another possible future upgrade, and that is some implementation of firewall.

A firewall should be implemented in JADEGateway. The idea is that not all the agents are prepared to accept unexpected messages from the outside of MAS and the whole computation could be disrupted if an unsuspecting agent were to receive an iESB message. Each MAS would define which agents can be contacted and only those would be accessible, thus creating a white-list of agents. This also deals with the issue of iESB to Jade discovery since available agents would be known.

White-listed agents would be prepared for iESB traffic and MAS disruption would be better prevented.

## 4. Database service

This document describes the ARUM database service specification.

### 4.1. Analysis

The ARUM database should provide data storage for other components of the ARUM system connected to iESB. Services deployed on iESB should be able to store, access and share data using the ARUM database. This should also be the preferred method of sharing large amounts of data between two or especially more than two services to prevent overuse of the underlying iESB message delivery system. The ARUM database is also useful for storing iESB service data during its life-cycle. This also holds for multiple deployments of the same service.

The ARUM database will be represented as an iESB service called the ARUM Database Service (ADS).

ADS should provide an interface to other services and using this interface, iESB services will be able to add, delete, change or read the database data or structure.

Additionally the data stored on ADS by a service shouldn't interact with different data of a different service if not intended to. It should be possible to isolate the data of any two services from each other. For this reason the ADS should provide access to multiple isolated database segments of various types (e.g. key-value store, column database). Each of these databases should be defined by their name, type and by their access rights. This bundle will be referred to as "database space" or short "dataspace".

Database spaces should represent sets of data of the same format and purpose accessible by only a limited number of services on iESB. In case that only single service is supposed to access a given database space, this database space will be called private. Other database spaces will be called public and can be accessed by multiple services.

The decision about which services can access which database spaces will be defined by the developers of the services and confirmed by the administrator of the iESB system.

The Database service should serve as a façade for different types of storage engines (e.g. key-value store or relational database) and provide a universal API for manipulation with them. The database plugin mechanism should be available to make it easy to add various database engines.

ADS should also provide a way to allow iESB services to watch parts of the database without active polling. This should be done using listeners, that would be possible to attach to a database and announce events in the database to the owner of the listener. This protocol is known as publish-subscribe.

#### 4.1.1. Use cases

This chapter enumerates scenarios in which ADS will be used. These use-cases provided a basis for the solution's design.

- UI preferences. In this scenario, a service or component that is used by human users can define UI preferences for each user. UI preferences generally differ from

case to case. A service that users work with can use ADS to store UI information. Each UI-enabling service can require a single persistent private database space.

- Security service data. A security service provides authorization and authentication of users working with services on the iESB. Rather than taking care of its own data, a security service could use the ADS for storing the information about users, passwords and authorizations.

A security service can require one or more private database spaces, additionally these database spaces will possibly need to be encrypted or otherwise secured in case any security breach happens.

- Logs. In this scenario, we have multiple services that wish to store log information into the database.

A log stores information that doesn't usually need to be accessed by the service that created it. Additionally, another service will need occasional access to this data in order to analyze the logs.

In this case a single public database space will be needed, accessible by the owner and the service that analyzes the data.

- Sniffer data An ARUM iESB Sniffer is an iESB component that logs traffic on ESB.

This scenario is similar to the logging scenario, however a sniffer might need to access the data that it creates.

Another service will need access to the data for their analysis and aggregation.

A sniffer will require a single table, since the only data type stored is ARUMMessage log information. This scenario doesn't need dynamic table creation and is independent on the user. A single Public Database space is needed, possibly a second table for aggregated values, accessed by the owner and the analyzing service or just the owner (the Sniffer).

- Service configuration Each service may store its settings in its own database space. These settings are now shared between all instances of that service and they are persistent between deployments and undeployments. This requires only a single private database space, accessed only by the owner service/services.

Databases integrated into ADS can be NoSQL [20] or SQL [21], ADS must be able to support both of those types and provide an interface to access them to iESB services.

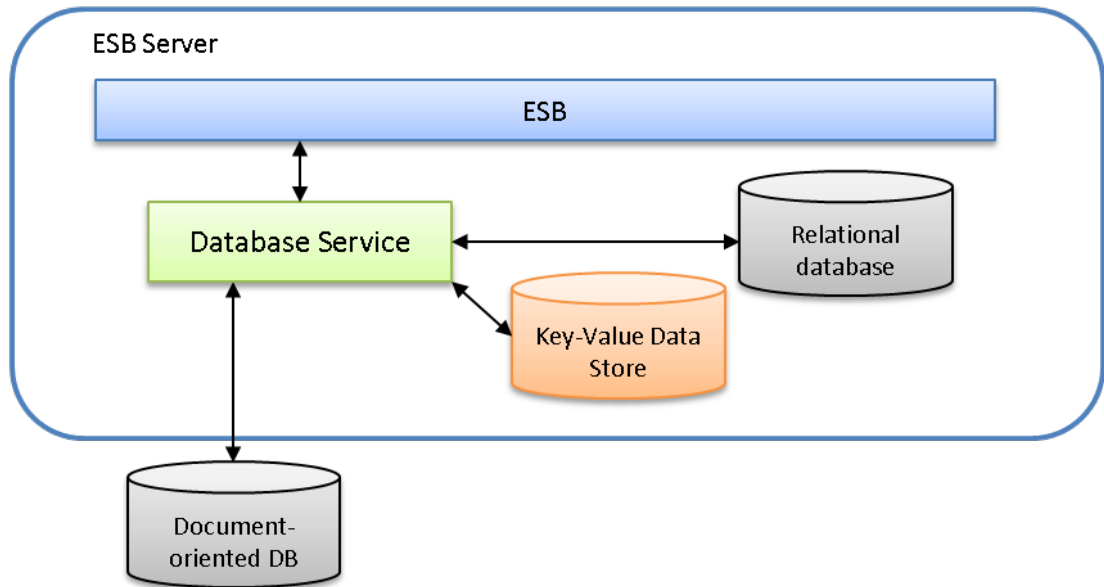
## 4.2. Architecture

The ARUM database service is an iESB service that integrates multiple ways of storing data for iESB disposal. It provides an abstraction over the connected database engines in the form of `databaseSpaces`. A dataspace represents a place where clients (iESB services) can store data. Each specific database engine that is plugged into the ADS must provide a mapping of database spaces to concrete database engine specific objects (e.g. a dataspace can be a single database in the relational [22] database engine or just a single table in case of column databases [20]). Each dataspace is identified by its name and type (e.g. key-value, relational, document etc.).

There can be multiple identical database engines connected to the ADS. Such a database engine is said to be present in multiple instances. For example, there can be three instances of Voldemort database version 0.9.1 connected to the ADS.

Figure 8 shows the basic architecture of ADS in ESB.

Figure 9 shows the internal structure and how ADS requests are processed. This process is independent on the type of database that is queried.



**Figure 8.** Database service architecture [23].

First, a message originates from the iESB, it has the correct format for ADSRequest, described in 4.4.1. Properties of ADSRequest important for parsing are `dataspaceName`, `databaseType`, `sender` (iESB service) and the inner content that encodes the request itself.

The message arrives to the ADS service, and it goes to the dataspace directory. The functionality of `DataspaceDirectory` is described in 4.6.

`DataspaceDirectory` executes the following checks:

- Directory checks whether either a private or a public dataspace of the requested `dataspaceName` exists. If it doesn't, the service returns a `DataspaceDoesNotExist` error.
- The ID of the sender service is compared to authorizations in the retrieved dataspace. If the service doesn't have any permissions for the dataspace, the service returns a `NotAuthorized` error.

As a next step, ADS transforms the request according to the dataspace that was retrieved. This, depending on database type, can mean renaming of the key names or table names using the dataspace prefix. This way no two dataspaces can tamper with others' data.

After the query was transformed, it is handed to a `PlugableDatabase` that holds the dataspace. The reference to the `PlugableDatabase` is defined in the `Dataspace` class.

After this point, operations are `databaseType` specific, generally speaking, the plug database transforms the request yet again and applies it on the underlying database engine that contains actual data from the dataspace. The plug database resolves the request and returns a matching response. Response matching is described in 4.4.1.

The response is then transformed back from the dataspace context and returned to the sender.

This process assures separation of various dataspaces, unifies the querying protocol and makes various databases available for iESB.

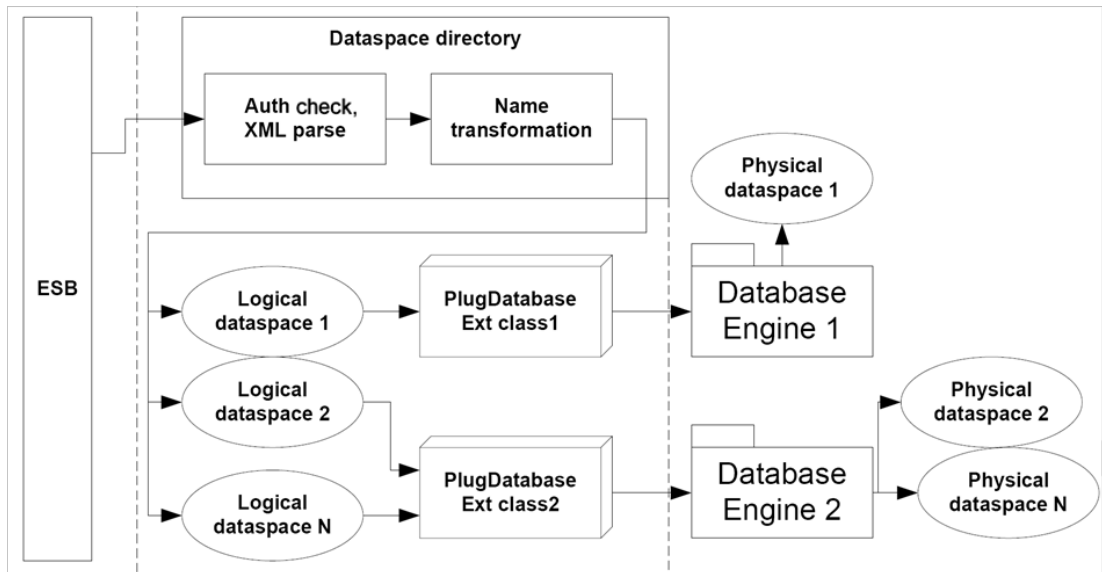


Figure 9. Architecture of request processing pipeline[23].

#### 4.2.1. Pluggable databases

A pluggable database is a pair of a database engine and a controller class that provides an API to incorporate the database into ADS. Attaching the controller to ADS allows it to communicate with iESB.

The controller class that connects some database engine must extend the PluggableDatabase interface, such a class will be referred to as a `PluggableDatabase`.

Figure ?? shows `PluggableDatabase` interface. It contains start/stop procedures for startup and shutdown, init procedure that is run after creation and before start and one procedure of each `OperationType`. When the request is parsed, it is split by `OperationTypes` and the parts of the request are passed to the operation(Add/Set/...) procedures in `PluggableDatabase`.

`PluggableDatabase` contains `ListenerManager` that takes care of dataspace listeners, settings and set of database spaces.

Code 4.1 Part of Pluggable database interface

```
public abstract class PluggableDatabase {
    protected ListenerManager listenerManager;
    protected Map<String,String> settings;
    protected Map<String, Dataspace> dataspace;

    public abstract DatabaseType getDatabaseType();

    public abstract boolean start();
    public abstract boolean backup();
    public abstract boolean stop();

    public abstract boolean isStarted();
    public abstract String add(String req);
    public abstract String set(String req);
}
```

```

    public abstract String put(String req);
    public abstract String get(String req);
    public abstract String delete(String req);
    protected abstract void init() throws Exception;
...}

```

### 4.2.2. Dataspaces

A database space represents a chunk of data stored on one of the underlying database engines. Database spaces are virtualized for use of iESB services. A single database space can be as of now stored only on one underlying database engine, multiple database spaces can be stored on a single database engine.

A database space is defined by the class `Dataspace`. Figure 4.2 shows an example of a dataspace interface.

A database space can be either public or private. Private dataspace allow easier coding of services that use them since from a service point of view, they always use the same dataspace.

Public database spaces do not belong to a particular service. Any service can request permissions to create and manipulate these spaces upon deployment. In this case, it is the System administrator who decides (during the deployment) to trust a service's need for access to a certain database space. Unlike private dataspace, public dataspace are independent on the service that created it. For example, if a service named "Serv" of category "Cat" creates a public dataspace named "Data", then the internal name of the dataspace will be "Data". Creation of dataspace is described in 4.5[23].

A private database space is only accessible to services sharing the same category and name (Which means it is shared only among different versions of the same service). In the CRUD model, this means that the service that created the private database space can perform all of the data CRUD operations and other services are not allowed to access the space at all. Private database spaces can additionally be either persistent or temporary. Persistent dataspace will not be generally deleted whereas temporary spaces can be deleted upon undeployment of the service. This is decided by the System administrator at the time the service is being undeployed. Private database space names use the name of the owner service as a prefix. This way multiple services can have a database space of the same name without interfering with each other. For example, if a service named "Serv" of category "Cat" creates a private dataspace named "Data", then the internal name of the dataspace will be "Cat\_Serv.Data"<sup>1</sup>[23].

Code 4.2 Part of Dataspace class interface

```

public class Dataspace {
    private HashMap<String, ServiceAuthorization>
        serviceAuths;
    private List<ADSListen> listeners;
    private DatabaseType type;
    private String dataspaceName;
    private String prefix;
    private Schema schema;
    private PluggableDatabase plugDatabase;
    private ARUMServiceEndpoint owner;
}

```

<sup>1</sup>Prefix formatting may change in future versions



```

    private boolean _private;
    ...
    public CfgDataspace getConfiguration() {...}
}

```

A dataspace is either created from backup, after ADS is restarted from shutdown or crash, or it is created from a configuration string when it's new, either way using JAXB[24]. JAXB serialization of the `Dataspace` class is described in detail in 4.3.

The following section describes the functionality of all the fields shown in 4.2:

- **HashMap<String, ServiceAuthorization> serviceAuths:** Each `Dataspace` stores information about what services are authorized to access it. Authorizations are described in more detail in 4.2.3. A dataspace holds a set of `ServiceAuthorizations` indexed by a string created from the unique address of the service.
- **List<ADSListen> listeners:** A `Dataspace` holds information about what active listeners are listening on the dataspace. More about listeners can found in 4.2.4.
- **DatabaseType type.** Database type of the dataspace. The `DatabaseType` of a dataspace must be the same as the `DatabaseType` of the `PluggableDatabase` in which the dataspace is nested.
- **String dataspaceName.** Name of the dataspace.
- **String prefix.** The prefix of the dataspace keys and table names. A prefix is created from the name of the dataspace and extended by name of the owner service if the dataspace is private. Prefixes are used to separate data from different dataspaces. A dataspace prefix is always unique. For example the same table used in two different dataspaces will not have the same name after applying a dataspace prefix to the name of the table.
- **Schema schema.** A schema is a `DatabaseType` dependent configuration of the database structure and it defines the internal structure of the dataspace. Contents vary depending on the type of the database.
- **PluggableDatabase plugDatabase.** A reference to the `PluggableDatabase` that owns the dataspace.
- **ARUMServiceEndpoint owner.** In case the dataspace is private, the owner property holds the address of the owner service. Otherwise it is null.
- **boolean \_private.** Whether the dataspace is private.

### 4.2.3. Authorization

Authorizations are defined using the `Authorization` class. Authorization define 3-level CRUD [25] for a single service: data, scheme and dataspace.

Data authorization level is the most used level of authorization. It allows data manipulation and it is the usual authorization level needed during runtime. Operations in data CRUD do not change the schema or any other global aspect of the database spaces.

Dataspace and scheme authorization levels are usually used during service deployment and undeployment. Scheme level allows change of scheme (adding and removing tables, adding columns to the tables etc.) and dataspace level allows manipulation with the dataspace (Dataspace creation and removal, authorization management etc.), as well as the management of underlying `PluggableDatabases`.

Each level is represented using a single number, CRUD operations are masked into the number:

- **0x1** for Create, allows a service to create objects that don't exist.
- **0x2** for Read, allows a service to query and read existing objects.
- **0x4** for Update, allows a service to edit existing objects.
- **0x8** for Delete, allows a service to delete existing objects.

The number created as a sum of the appropriate masks represents the authorization.

Figure 4.3 shows a preview of the `Authorization` class fields.

Code 4.3 Part of `Authorization` class interface

```
public class Authorization {
    protected int dataCrud;
    protected int dataspaceCrud;
    protected int schemeCrud;
    protected String serviceCategory;
    protected String serviceName;
    ...
}
```

Functionality of `Authorization` is dependent on the implementation of the security component in the ARUM system.

As of now, services that communicate over iESB using `ARUMMessages` must fill in the `From` property of `ARUMMessage`. `From` is an `ARUMServiceEndpoint` class instance, or in other words, a descriptor of a single service on iESB. However nothing forces the service to fill in an `ARUMServiceEndpoint` matching the service. A service can pretend it is a different service and that way it can access restricted dataspace that should be inaccessible.

An `ARUMSecurity` component is in development and it will make ADS authorization secure. When `ARUMSecurity` is deployed on iESB, services won't be able to pretend they are a different service.

#### 4.2.4. Listeners

Listeners are introduced to allow some services to observe certain data and wait for a change without the need of polling. For this reason a service can be registered to listen to changes and events in a given dataspace. The `ADSListen` class stores the configuration of such a listening registration.

The `ADSListen` class is based on the XML format shown in chapter 4.3 (figure 4.5).

`ADSListen` class instances are shared between a `Dataspace` and its `PlugableDatabase`, a set of all `ADSListen`s in all `Dataspace`s in a single `PlugableDatabase` corresponds to a set of all of its `ADSListen`s.

During each request, for each operation in the request, if the operation is successful, `ADSListen`s are queried for those who match in dataspace and operation type. Afterwards, a set of services from each operation is sent a message. The class that takes care of resolving listening is `ListenerManager` 4.6 and each `PlugableDatabase` contains a single instance to resolve its listeners.

#### 4.2.5. Database types

This section shows supported or in future to be supported noSQL/SQL database engine types. Database engines in a single type don't necessarily share the same interface. It only means they are similar enough for them to be accessed using the same protocol.

Adding a new database engine to the portfolio of an ADS service requires these steps.

1. **Creation of PlugDatabase class.** The developer must create a new class that implements the `PlugableDatabase` interface. This class converts unified ADS protocols of data manipulation and scheme/dataspace creation into the native language/protocol of the database engine that is to be integrated. At this point, the developer chooses one of the existing database types and besides general ADS protocols she implements a database-type-specific XML parsing or uses JAXB classes that are available in the ADS library.
2. **Configuration.** To deploy a new `PlugableDatabase` to the ARUM iESB, the administrator must add a reference to the `PlugableDatabase` into the startup configuration file of the ADS.
3. **Database management.** If the `PlugableDatabase` doesn't contain direct manipulation with database engine, the administrator must manually create and maintain the newly added database engine.

### Key-value dataspace type

Key-value databases are defined in a noSQL format [`keyvalnosql`]. Two implementations of key-value **PlugableDatabases** were created for ADS and this thesis: the Voldemort key-value database [26] and the FoundationDB database [27].

A key-value database contains a set of key-value pairs, in some implementations each pair has an incrementing version number. Both the key and the value are represented as strings. It is the simplest type of database available in the noSQL format.

The XML code is in the Appendix section in A.1. A short example is in the figure 4.5.

A key-value database space uses only a single data format, which is the XML element `entry`. Additionally, a set of `entries` can be encapsulated inside the tag `entry-range` or `entry-key-range`.

Querying multiple entries is done by querying all entries that have keys that share a prefix. That is executed using the `entry-range` or `entry-key-range` tags. These two tags represent sets of `entries` that share the same prefix defined in the attributes of the tag. For example `<entry-range key="st">` tag covers any keys in the form "st\*", for example "start" or "station".

If an `entry` tag is inside `add`, `put` or `set` tags, the database tries to add the key-value to the underlying database. If it's used inside `get` or `delete`, the database only needs the `key` attribute and it either returns the value or fills the `success` attribute and returns the tag.

The `entry-range` and `entry-key-range` tags both contain sets of `entries`, however in the get operation the `entry-key-range` returns only the keys of the queried data and leaves values blank.

Code 4.4 Request/response example of data manipulation protocol in key-value case

```

<put>
    <entry key="MyKey" value="ThisIsData"/>
</put>
<get>
    <entry-range key="Key"/>
    <entry-key-range key="Key"/>
</get>
Response nodes:
<put>
    <entry key="MyKey" success=true version=2/>

```

```

</put>

<get>
  <entry-range key="Key">
    <entry key="Key1"
      value="ThisIsData" version=2/>
    ...
  </entry-range>
  <entry-key-range key="Key">
    <entry key="Key1"/>
    ...
  </entry-key-range>
</get>

```

The `entry-range` and `entry-key-range` tags can be only used in `get` and `delete` tags. The operation is applied to all the data in the dataspace and `entries` are returned inside of the tag for each successful hit.

### Column dataspace type

The column dataspace type is the second implemented dataspace type, for the thesis a single `PluggableDatabase` was implemented, and that is the `HBaseRemote` pluggable-Database, which implements the Hadoop HBase column database [28].

A column database is similar to a key-value database, only the "value" in case of the column database is a set of values. Each entry in a column database is a single key and a set of "columns", where each column has a key and a value (both strings). Available column keys must be defined when the dataspace is created. A single dataspace in ADS is represented by one or more tables with predefined columns. Each can contain none, one or more of these columns, there is no requirement to fill any columns besides key, which is used for identification. Some column noSQL databases have a more broad definition and a more general use, however in ADS we restrict ourselves to this format.

In a column database type, the equivalent to key-value's `entry` is the tag `row`. `Row` defines `key` attributes. The equivalent to key-value's `entry-range` is `row-range`. A `row-range` contains a set of `row` tags.

The XML element `row` contains a set of XML elements `col`. Each of the `col` elements represents a single column and its value, the key is stored in the attribute "cn".

XML code is in the appendix section in A.2. A short example is in the figure 4.5. In the example, a request asks to put a new row into the table named "tname", in the reply ADS returns that the operation was successful. Additionally a request queries the same table for all rows whose key starts with the prefix "", which is any row so the query returns all rows in the table.

Code 4.5 Request/response example of data manipulation protocol in key-value case

```

<put table="tname">
  <row key="MyKey">
    <col cn="col1">Value</col>
    <col cn="bcol">Value2</col>
  </row>
  ...
</put>

```

```

<get table="tname">
  <row-range key=""/>
</get >
Response nodes:
<put table="tname">
  <row key="MyKey" success="true/">
    ...
</put>
<get table="tname">
  <row-range key="">
    <row key="MyKey">...</row>
    <row key="AnotherKey">...</row>
  </row-range>
</get >

```

### 4.3. XML serialization

All messages used in ADS communication are formatted into XML. This format was chosen as the most widely used with a high amount of existing tools that work with XML, XSD and related formats. Since ARUM iESB is implemented in Java, an appropriate tool to transform XML data into Java structures was needed.

Eventually two tools were used. The primary method to transform XML to Java is JAXB[24]. The following parts of the ADS implementation use JAXB-generated classes:

- All of the database-type-specific operations are encoded in XML and have their Java classes stored in the package `cz.certicon.arum.ads.jaxb.*` and all the child packages.
- Classes that are used for configuration, backup serialization and some parts of Dataspace manipulation protocol. These classes are inside the `cz.certicon.arum.ads.config` package.

XSD files for each JAXB Java class can be found in the CD attached to this thesis.

This chapter shows examples of XML formats used in ADS messaging and class serialization for backups and configurations.

#### 4.3.1. Database event listening

4.6 An `ADS-listen` tag can be seen in dataspace configuration, during deployment of new dataspace or in backups that `PlugableDatabase` use to remember existing database spaces. It describes the relationship between one or more dataspace and an iESB service.

A dataspace acknowledges `services-to-contact` when the `dataspace` receives a request and executes an operation of one of the types in `operationtypes`.

Code 4.6 ADSListen XML example

```

<ADS-listen dataspace="MyDataspace" data-type="KeyVal"
  listener-id="id">
  <KeyNamePrefix prefix="Pref."/>
  <operationTypes>
    <opType type="Add"/>

```

```

    </operationTypes>
  <services-to-contact>
    <service name="ServiceName"
      category="ServiceCategory"/>
  </services-to-contact>
</ADS-listen>

```

### 4.3.2. Database schema

This XML is used in dataspace configuration, in dataspace manipulation protocol, configuration and backup synchronization. Figure 4.7 shows an example of schema configuration for a column database. A column database was chosen for this example because the key-value database type doesn't have any contents of the scheme.

Code 4.7 Dataspace scheme configuration XML example

```

<schema schema-id="columnDatabase1" version-major="1"
  version-minor="0">
  <table name="tableName">
    <col>Column1Name</col>
    <col>Another column</col>
    ...
  </table>
</schema>

```

### 4.3.3. Authorization

Authorization assigns 3 levels of CRUD to a single service. Authorization class/XML is one of the properties of requests and dataspaces. When a request is processed, each operation has a CRUD level. The `Authorization` class defines a set of functions that compare authorizations and it either refuses the access or allows the operation to be executed.

The XML element `Authorizations` is a set of `Authorization` elements, an example is shown in figure 4.8.

Code 4.8 Dataspace scheme configuration XML example

```

<authorizations>
  <authorization service-name="NameService"
    service-category="Category"
    data-crud="1111" />
  <authorization service-name="NameOfTheService"
    service-category="Category"
    data-crud="1110" />
</authorizations>

```

### 4.3.4. Dataspace

Figure 4.9 shows an example of XML configuration of a single dataspace.

The attribute `instance-id` does not have to be defined. If it is defined, it contains the ID of `PlugableDatabase`, which is the preferred choice for where the dataspace

will be nested. In case the ID is defined and no such `PluggableDatabase` with that ID exists, ADS returns an exception in deployment of the dataspace. If no `instance-id` is defined, ADS chooses one of the `PluggableDatabases` by itself.

The attribute `delete-on-undeployment` tells ADS to allow the undeployment process to ask for the deletion of the dataspace. This attribute can be true only in case that the dataspace is set as private.

Code 4.9 Dataspace scheme configuration XML example

```
<dataspace name="HbaseDS" data-type="column"
  private="false" delete-on-undeployment="false"
  instance-id="HBaseColumn">
  <authorizations>
    ...
  </authorizations>
  <schema schema-id="colSch" version-major="1"
    version-minor="6">
    ...
  </schema>
  <listeners>
    <ADS-listen dataspace="HbaseDS"
      data-type="Column" listener-id="id">
      ...
    </ADS-listen>
    ...
  </listeners>
</dataspace>
```

## 4.4. Protocols

This section shows communication protocols between ADS and other iESB services, a protocol for the management of ADS by administration services and procedures that are followed during ADS restarts, startups and after unexpected shutdowns.

### 4.4.1. Data manipulation protocol

This protocol represents data querying by services on iESB and it is the main purpose of ADS. This protocol is evoked by one of the iESB services.

The ADS strives to provide a unified interface for data manipulation across the possible physical databases on different levels. The first interface unification is applied on the data operations level. The following list of data operations (mapped to the data CRUD permissions) has been identified to cover the needs of any physical database:

- Add (Create): Inserts data item into the dataspace. Fails if the data item already exists.
- Put (Create, Update): Inserts data item into the dataspace. Overwrites existing data if it already exists.
- Set (Update): Sets value for the existing piece of data, fails if it does not exist.
- Get (Retrieve): Retrieves data from the dataspace. Fails if the data item doesn't exist.

- Delete (Delete): Removes data from the dataspace. Fails if the data item doesn't exist.

Each operation can additionally fail when the service is not authorized to execute them.

These operations are encoded in XML in the fields `OpType` and in the enum class `OperationTypes`.

Note that each database type might support only a subset of the listed data operation methods. The second level of interface unification is applied on the database type level. Each database type supports a single language for specification of the details of the data operations above. This means that the data manipulation language is common to all the possible underlying physical database engines of the same type.

Communication on iESB is always mapped into one of the FIPA protocols [1] and the data manipulation communication protocol is based on the FIPA-Query protocol. All requests for the database service come in the form of XML iESB `ARUMMessage`. Similarly, ADS replies with XML iESB `ARUMMessage`.

These two XML trees are isomorphic. Each request node in the request XML corresponds to a reply node in the response XML. The only difference is when one of the operations fail. In that case a subtree in the request may lead to a single node in the response. Similarly in the case of queries that return more than one item. In that case, a single node in the request may correspond to a set of nodes or a subtree in the response.

The request and response messages have 3 levels of XML elements[29]:

1. The Request root element identifying the database space name and its type. Type is here just for consistency check. A `Response` element is returned in the response message. It contains error details in case of a failure or XML matching in the structure of the request. Possible errors for dataspace level include:
  - `NoSuchDataspace`: the specified dataspace does not exist
  - `AccessDenied`: the database space exists but the requestor does not have permissions to access it. None of the CRUD operations are allowed for the service.
  - `TypeMismatch`: the specified database space type does not match the actual database space type.
  - `UnknownFatalError`: any other unexpected error.

These errors are encoded in the class and can be returned in an `<ADSRequest .../>` node.
2. Operation elements defining what operations need to be executed. A matching operation element is returned in the response. The response operation element contains error details in case of a failure of the operation as a whole. Possible errors on operation include:
  - `UnknownOperation`: the specified operation does not exist.
  - `OperationNotSupported`: the specified operation is not supported by the database type.
  - `AccessDenied`: the operation exists and is supported but the requestor does not have the permissions to execute the operation.
  - `UnknownFatalError`: any other unexpected error.
3. Operation details in the database-type-specific language. Multiple data items can be manipulated within a single Operation element. For example, multiple values can be added into a Key-Value database within a single add operation element. Errors on the data level are database-type-specific but may include:



- DoesNotExist: the data does not exist. This happens especially for Set, Get and Delete operation types
- AlreadyExists: the data already exists. This error happens only for the Add operation type.
- NewerVersion: the data already exists in a newer version. This error happens for dataspace nested in `PluggableDatabases` that support versioned data and it can happen for Set, Put and Delete operations.
- UnknownFatalError: any other unexpected error.

An example of a general request response message structure is in figure 4.10.

Code 4.10 Example of ADSRequest and ADSResponse xml structures

Request:

```
<ADS-request dataspace="MyDataspace" data-type="Key-value"
continue-on-failure="false">
  <add>
    ...operation details in database
    type-specific language...
  </add>
  <unknown-op>
    ...operation details in database
    type-specific language...
  </unknown-op>
</ADS-request>
```

Response elements examples:

```
<ADS-response dataspace="MyDataspace"
data-type="Key-value">
  <add>
    ... responses ...
  </add>
  <unknown-op error-code="7"
error="UnknownOperation">
    ... error details ...
  </unknown-op>
</ADS-response>
```

...or...

```
<ADS-response dataspace="MyDataspace" data-type="Key-value"
error-code="1" error="DoesNotExist">
  Description of the error if applicable
</ADS-response>
```

All operations inside the `request` element are executed sequentially. Execution order inside the `operations` elements is not guaranteed. For example if one `<add>` node contains multiple add requests, their execution can be parallel.

The `continue-on-failure` attribute specifies what happens in case an operation fails. If it is set to false, the processing of the message stops after the first error is encountered, otherwise it continues with the next operation.

FIPA failure performative[1] is returned only in case of an error on the root level element. All the other response messages are of the Inform-Result FIPA type even if some of the operations have not been successful.

Contents of the separate operations vary depending on database type, the implemented types are described in 4.3.

#### 4.4.2. Dataspace administration protocol

Dataspace administration protocol is similar to Data manipulation protocol. The Dataspace administration protocol also uses an XML format to encode requests and responds with a matching XML tree.

The administrative commands communication is based on the FIPA-Request protocol [1]. Administrative operations are available only to administration services like ARUMDeployer. Execution of these operations by any other service is prohibited. ADS supports the following list of administrative operations (mapped to the administrative CRUD permissions):

- **Request database space (Create):** Creates a database space or updates the existing database. The type of the database must be specified together with its access level (private/public) and the physical database instance in which the database space should be created. The Inform-done FIPA message is returned upon the successful creation of the database space. In case of a private database space, the deletion policy must be set. The Failure FIPA message is returned in case of an error. An example is shown in 4.11.

Code 4.11 Example of create request in DAP protocol

```
<ADS-admin function="create">
  <dataspace name="MyDataspace" data-type="Relational"
    private=true delete-on-undeployment=false
    instance-id="Mysql01">
    <authorizations>
      <authorization service-name="ServiceName"
        service-category="SCategory"
        data-crud="1111" />
    </authorization>
    </authorizations>
    <schema schema-id="logSchema" version-major="1"
      version-minor="6">
      ...Database type dependent schema
      definition...
    </schema>
  </dataspace>
</ADS-admin>
```

- **List Available Database Spaces (Retrieve):** Retrieves the list of existing database spaces based on their types. An example is shown in 4.12.

Code 4.12 Example of get request in DAP protocol

Example request that lists all database spaces:

```
<ADS-admin function="get-dataspaces"/>
```

#### 4. Database service

Example request that lists key-value database spaces:

```
<ADS-admin function="get-dataspaces">
  <space-type name="key-value" />
</ADS-admin>
```

Example of response:

```
<ADS-admin function="get-dataspaces">
  <space-type name="key-value">
    <dataspace name="Dataspace1" data-type="key-value"
      private="false"
      instance-id="Voldemort001"
      delete-on-undeployment=false />
    <dataspace name="Dataspace2" private="true"
      owner="Cat.Serv"
      instance-id="Voldemort001" />
  </space-type>
  <space-type ...>
    ...
  </space-type>
</ADS-admin>
```

- **Get database space details (Retrieve):** Retrieves detailed information about a database space. The Inform-result FIPA message is returned in case the operation is successful. The Failure FIPA message is returned in case of an error. An example is shown in 4.13.

Code 4.13 Example of get-details request in DAP protocol

Request:

```
<ADS-admin function="get-dataspace-schema">
  <dataspace name="Dataspace"/>
</ADS-admin>
```

Response gives list of existing dataspace:

```
<ADS-admin function="get-dataspace-schema">
  <dataspace name="MyDataspace" data-type="Relational"
    private=true
    delete-on-undeployment=false
    instance-id="Mysql01">
    <authorizations>
      <authorization service-name="HappyService"
        service-category="NiceServices"
        data-crud="1111" />
    </authorization>
    </authorizations>
    <schema schema-id="logSchema" version-major="1"
      version-minor="6">
      ...Database type dependent schema
      definition...
    </schema>
  </dataspace>
```

```
</ADS-admin>
```

- **Remove database space (Delete):** Removes a single database space. The Inform-done FIPA message is returned upon the successful deletion of the database space. The Failure FIPA message is returned in case of an error. An example is shown in 4.14.

```
Code 4.14 Example of delete request in DAP protocol
```

Example Request:

```
<ADS-admin function="delete">
  <dataspace name="DataspaceName"/>
</ADS-admin>
```

- **Get Available Database Space types (Retrieve):** Retrieves a list of available physical database engines together with their instances and their types. Each of the returned entries represents an active PlugableDatabase. An example is shown in 4.15.

```
Code 4.15 Example of PlugableDatabase query request in DAP protocol
```

Request:

```
<ADS-admin function="get-dataspace-types" />
```

Example response:

```
<ADS-admin function="get-dataspace-types">
  <space-type name="key-value">
    <database-impl name="Voldemort" version="0.96"
      class-name="PlugDatabaseVoldemort">
      <database-instance id="Voldemort001"
        name="PrimaryKV" />
      <database-instance ...>
        ...
      </database-instance>
    </database-impl>
  <database-impl ...>
    ...
  </database-impl>
</space-type>
<space-type ...>
  ...
</space-type>
</ADS-admin>
```

### 4.4.3. ADS lifecycle

This section explains the processes that take place during an ADS lifecycle.

#### Startup

1. **ADS initialization.** The ADS starts on JBoss AS, inside iESB. When the ADS is initialized, it sets up all internal structures used for all other components.
2. **Configuration loading** During initialization, ADS loads XML contents of the file `./META-INF/dataconfig.xml`. This file contains ADS service configuration,

configuration of all `PlugableDatabases` that need to be started in ADS and all dataspace that must be deployed.

3. **Plugdatabases initialization** `PlugableDatabase` configurations are loaded in the previous step. ADS goes through each of them and starts the databases. Each database usually carries its own dataspace that were deployed previously. ADS registers these dataspace and revives them. Data are preserved as long as the database is persistent.
4. **Dataspace load.** More dataspace can be loaded from the configuration and it gets initiated and deployed on the already initialized `PlugableDatabases`.

During the addition of a dataspace to the dataspace registry on ADS, all of the listeners in the `Dataspace` configuration are registered in the assigned `PlugableDatabase`.

#### **New dataspace backup**

After a dataspace was deployed onto a `PlugDatabase`, the `PlugDatabase` calls the dataspace backup procedure. XML configuration is extracted from existing dataspace inside the `PlugDatabase`, including schema and listeners. All of this configuration is stored inside the database itself as a string. It is stored in a location in the database that no other service can access through dataspace.

Details are found in 4.3. If ADS is halted and restarted, each `PlugDatabase` loads its own dataspace and announces them to the ADS.

### **4.5. Deployment**

The creation of a database space and updates to its schema (should the database require one) is restricted to iESB service deployment only. Each iESB service defines in its deployment package what database space it requires. Each database space structure should be predefined outside ADS and deployment process. Services sharing a single database space should have this knowledge. In case the database space structure needs to be updated, it can be updated either with backwards compatibility or without it. In case it has backwards compatibility with an older version, it shares the same major version and increments the minor version.

Versioning is done in the following manner:

1. If a database space does not exist, its version is 0.
2. If a service that is being deployed uses the same major version of database space as the version currently in place, the deployment can continue.
  - a) In case the minor version is older, no action is taken.
  - b) In case the minor version is newer, the database space is updated according to the new schema.
3. If the service that is being deployed uses an older major version then the deployment is halted and the developer of the service must update it to be able to work with the newer version.
4. If the service that is being deployed uses a newer major version and there are services using an older version deployed:
  - a) Deployment is halted
  - b) The System administrator must decide either:
    - i. Not to deploy the new service or
    - ii. To undeploy all services using older major version.

- If the service that is being deployed uses a newer major version and there are no services using an older version deployed, the database space is updated to the new schema version and the service is deployed.

Figure 10 shows an example of dataspace versioning succession. The left side shows new deployments that request the same dataspace. The right side shows actions that are taken (Create, OK, Update) and the new state of the dataspace (in green) and information about authorizations (in blue). The last action in the succession shows that an update of the dataspace to a newer major version needs the undeployment of all services that use the older dataspace version. This prevents data splitting by version and keeps the management of the data simple.

The standard procedure of upgrading the major version starts with one of the services updated to use a newer major version. Deployment fails because of reasons mentioned and the administrator would have to undeploy all services using the old version, which she won't do. Deployment will be delayed until all of the services are updated to use the newer major version and after that all the services are redeployed and the dataspace is updated.

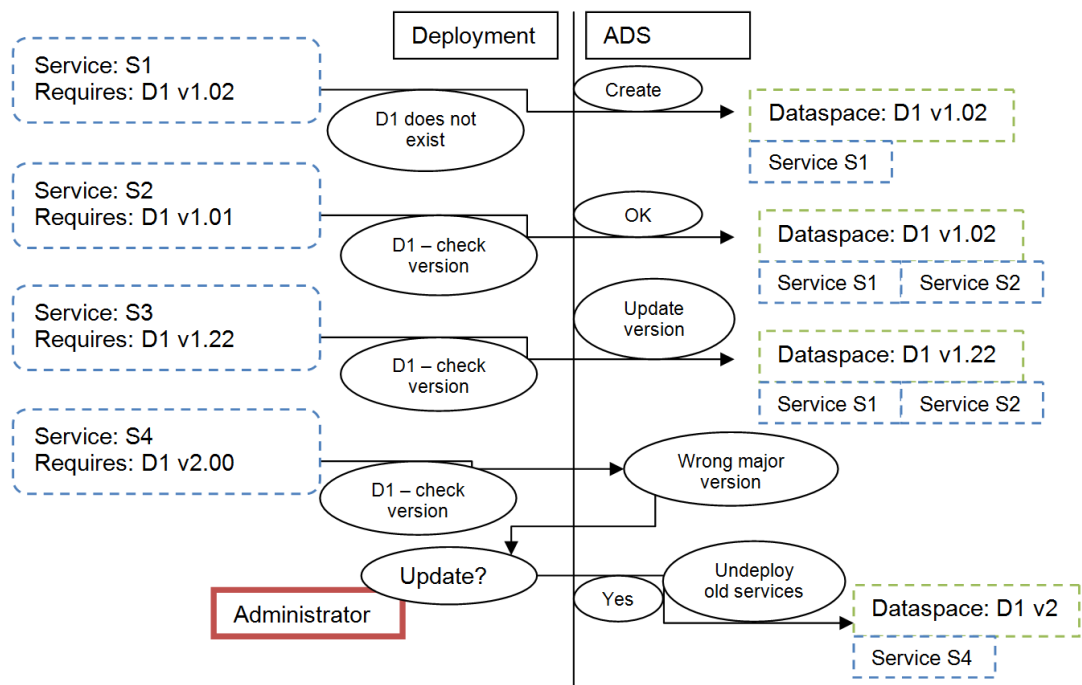


Figure 10. Dataspace versioning succession example [23].

## 4.6. Other interfaces and classes

- **ADSRequest** and **ADSResponse** classes contain functions to recognize work with ADS-request XML format. They implement ARUMMessage interface, but in fact they contain the original message.
- **Query processor classes** Query processors are used when transforming request nodes from dataspace independent format to dataspace dependent format and after execution the other way around. This usually concerns formatting and adding/removing dataspace prefix to/from key/table names. **iltemListenerManager** is a

class that indexes listeners assigned to a single `PlugableDatabase`. It manages addition and removal of `ADSListener` classes as well as detecting listening event. Listeners are `DatabaseType` specific and one class has to be implemented for each type.

- **XMLHelper** XMLHelper is a class that contains set of static procedures that help manual parsing of XML. It uses DOM XML parser implementation. Code can be seen on CD.

### 4.7. Future work

ADS database is still in development.

One of the possible future features can be Schema comparison algorithm, that will help the developers and administrators make schema updates easier and with less configuration.

ADS will be extended by support for more database types. Those will probably be noSQL document databases and old fashioned SQL relational database. Each database type should have at least single `PlugableDatabase` implemented

Another possible feature that was discussed in circles of Certicon development team was an idea to speed up large data transfer by allowing the client service access the underlying databases directly without all the transformations during ADS protocols.

This could highly speedup the process, only meta-data would be transferred through iESB and the actual data would go directly using standard TCP/IP over the network. Sniffer could still track the traffic since the bypass initialization was still done over iESB.

Problem here is possibility of database inconsistency, when uncontrolled access or access not controlled enough would make a mess inside the ADS databases. This would be security risk.

Users of iESB might want to encrypt some very sensitive data. This can be case of SecurityService data and there are plenty scenarios, where data should be encrypted. As one of the next features, ADS could support encrypted dataspaces. Ideally to design asymmetric architecture, where keys are kept at the client service, however ADS could be able to decrypt part of the data to apply the standard ADS functions.

ADS doesn't support advanced aggregation functions. This should and could be easily implemented in short amount of time.

## 5. Conclusions

In conclusion, all three components were successfully implemented. For each tool or library, problem was analyzed and architecture was derived and implemented in straight forward and universal manner.

ARUM project is still in heavy development and all of tools that are described in this thesis will undergo changes towards the end of the project. Each tool can be extended and optimized.

ADS enabled iESB deployment independent data storage that supports multiple implementations of key-value and column/table databases. When administration part implemented in ARUMAdminService is finished, the whole process of managing the ADS will be automated and executed during deployments. Until then, ADS can still be functional if configured manually.

ARUMJadeGateway is the most mature project of the three tools demonstrated. It is already being used in other components as a library and probably won't change much besides suggestions shown in future work section. Jade platforms on iESB can now communicate and use all the capabilities of the iESB and MAS functionality can now be invoked directly from iESB, without repeatedly implementing MAS front-end.

ARUMDeployer provides core for deployment system that will be used in ARUM release. It is under heavy development and it will be part of ARUM administrator toolbox. For now it allows remote deployment and undeployment of iESB tools using just web browser.

XSD and XML files, that make up large portion of the ARUMDatabaseService implementation are on CD.



# Appendix A.

## Appendix

### A.1. ARUM database service XML/XSD examples

#### A.1.1. Data manipulation XML

Code A.1 Request/response example of data manipulation protocol in key-value case

```
<ADS-request dataspace="MyDataspace" data-type="Key-value"
continue-on-failure="true">
  <add>
    <entry key="MyKey" value="ThisIsData"/>
  </add>
  <put>
    <entry key="MyKey" value="ThisIsData"/>
  </put>
  <get>
    <entry key="MyKey"/>
  </get>
  <delete>
    <entry key="MyKey"/>
  </delete>
  <get>
    <entry key="MyKey"/>
  </get>
  <get >
    <entry-range key="Key"/>
  </get>
  <get >
    <entry-key-range key="Key"/>
  </get>
</ADS-request>

<ADS-response>
  <add>
    <entry key="MyKey" success=true version=1 />
  </add>
  <put>
    <entry key="MyKey" success=true version=2/>
  </put>
  <get>
    <entry key="MyKey" value="ThisIsData" version=2/>
  </get>
</ADS-response>
```

```

</get>
<delete>
  <entry key="MyKey" success=true version=2/>
</delete>
<get>
  <entry key="MyKey" error-code="1" error="DoesNotExist"/>
</get>
<get >
  <entry-range key="Key">
    <entry key="Key1" value="ThisIsData" version=2/>
    <entry key="Key25" value="DataThisIs" version=6/>
  </entry-range>
</get >
<get >
  <entry-key-range key="Key">
    <entry key="Key1"/>
    <entry key="Key25"/>
  </entry-key-range>
</get >
</ADS-response>

```

Code A.2 Request/response example of data manipulation protocol in column case

```

<ADS-request dataspace="MyDataspace" data-type="KeyVal">
  <add table="name">
    <row key="MyKey">
      <col cn="colN">Value</col>
      <col cn="bcol">Value2</col>
    </row>
    <row key="MyKey" >
      <col cn="colN">Value</col>
      <col cn="bcol">Value2</col>
    </row>
  </add>
  <put table="name">
    <row key="MyKey">
      <col cn="colN">Value</col>
      <col cn="bcol">Value2</col>
    </row>

    <row key="MyKey">
      <col cn="colN">Value</col>
      <col cn="bcol">Value2</col>
    </row>
  </put>
  <set table="name">
    <row key="MyKey">
      <col cn="colN">Value</col>
      <col cn="bcol">Value2</col>
    </row>

```

```

    <row key="MyKey">
      <col cn="colN">Value</col>
      <col cn="bcol">Value2</col>
    </row>
  </set>
  <get table="name">
    <row-range key="">
      <row key="MyKey"/>
      <row key="MyKey"/>
    </row-range>
    <row-range key="">
      <row key="MyKey"/>
      <row key="MyKey"/>
    </row-range>
    <row key="MyKey"/>
    <row key="MyKey">
      <col cn="colN">Value</col>
      <col cn="bcol">Value2</col>
    </row>
  </get>
  <delete table="name">
    <row-range key="">
      <row key="MyKey"/>
      <row key="MyKey"/>
    </row-range>
    <row-range key="">
      <row key="MyKey"/>
      <row key="MyKey"/>
    </row-range>
    <row key="MyKey"/>
    <row key="MyKey"/>
  </delete>
</ADS-request>

```

# Appendix B.

## CD contents

```
ROOT
...\thesis.pdf
...\projects\
...\...\ARUMDeployer
...\...\projects\ARUMJadeGateway
...\...\projects\ARUMDatabaseService

...\configurations\
...\...\ARUMDeployer
...\...\projects\ARUMJadeGateway
...\...\projects\ARUMDatabaseService
...\...\...\XML
...\...\...\XSD

...\other
...\...\documents
```

## Bibliography

- [1] IEEE Computer Society. *The Foundation for Intelligent Physical Agents*. 2014. URL: <http://www.fipa.org/>.
- [2] Ing. Pavel Vrba PhD. *ARUM general presentation*. 2013.
- [3] Luis Ribeiro, Jose Barata, and Armando Colombo. "MAS and SOA: A case study exploring principles and technologies to support self-properties in assembly systems". In: (2008), pp. 192–197.
- [4] Jeffrey D. Ullman. "< i> NP</i>-complete scheduling problems". In: *Journal of Computer and System sciences* 10.3 (1975), pp. 384–393.
- [5] Airbus. *Airbus.com*. 2014. URL: <http://www.airbus.com/>.
- [6] Airbus.com. *A380 production map*. 2014. URL: <http://cdn.grin.com>.
- [7] David A Chappell. *Enterprise service bus*. " O'Reilly Media, Inc.", 2004.
- [8] JBoss. *Jboss SOA solution*. 2014. URL: <http://jbossesb.jboss.org/>.
- [9] JBoss. *JBoss messaging, HornetQ*. 2013. URL: <http://www.jboss.org/hornetq>.
- [10] Richard Sezov. *Liferay Administrator's Guide*. Lulu. com, 2008.
- [11] Jason Brittain and Ian F Darwin. *Tomcat: the definitive guide*. " O'Reilly Media, Inc.", 2007.
- [12] Mukesh Singhal. "Deadlock detection in distributed systems". In: *Computer* 22.11 (1989), pp. 37–48.
- [13] Joxan Jaffar and J-L Lassez. "Constraint logic programming". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1987, pp. 111–119.
- [14] Ing. Martin Klima PhD. *ARUM introduction*. 2013.
- [15] Ing. Tomas Budin. *ARUM screenshots for deliverables*. 2014.
- [16] W3. *HTTP*. 1988. URL: [www.w3.org/Protocols](http://www.w3.org/Protocols).
- [17] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. "JADE–A FIPA-compliant agent framework". In: *Proceedings of PAAM*. Vol. 99. 97-108. London. 1999, p. 33.
- [18] F Laburthe and N Jussien. "Choco solver documentation". In: *Ecole de Mines de Nantes* (2011).
- [19] Dominic Greenwood et al. "The IEEE FIPA approach to integrating software agents and web services". In: *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. ACM. 2007, p. 276.
- [20] Jing Han et al. "Survey on NoSQL database". In: *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE. 2011, pp. 363–366.
- [21] Michael Stonebraker. "SQL databases v. NoSQL databases". In: *Communications of the ACM* 53.4 (2010), pp. 10–11.

## Bibliography

- [22] David Maier. *The theory of relational databases*. Vol. 11. Computer science press Rockville, 1983.
- [23] Bc. Michal Fuksa. *ARUM Central database*. 2013.
- [24] Ed Ort and Bhakti Mehta. “Java architecture for xml binding (jaxb)”. In: *Sun Developer Network* (2003).
- [25] Wikipedia. *CRUD*. 2014. URL: [https://en.wikipedia.org/wiki/Create,\\_read,\\_update\\_and\\_delete](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete).
- [26] Jing Han et al. “Survey on NoSQL database”. In: *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE. 2011, pp. 363–366.
- [27] Lawrence M Rausch. *Project Officer. Information Tech, nology Innovation Survey: Fall 2001, National Science Foundation [DB ?]* 2004.
- [28] Mehul Nalin Vora. “Hadoop-HBase for large-scale data”. In: *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*. Vol. 1. IEEE. 2011, pp. 601–605.
- [29] Bc. Michal Fuksa. *ARUM ADS database*. 2013.
- [30] D. Šišlák et al. “AGENTFLY: Towards multi-agent technology in free flight air traffic control”. In: *Defence Industry Applications of Autonomous Agents and Multi-Agent Systems* (2008), pp. 73–96.
- [31] John Koenig. “JBoss jBPM”. In: *White Paper, November* (2004).