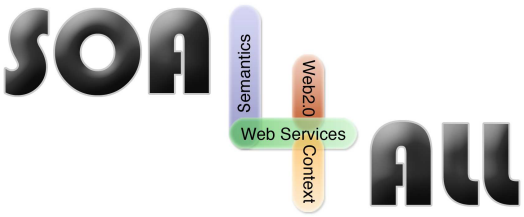


Project Number: **215219**

Project Acronym: **SOA4All**

Project Title: **Service Oriented Architectures for All**

Instrument: **Integrated Project**

Thematic Priority: **Information and Communication Technologies**

D1.4.1B SOA4All Runtime

Activity N:	Activity 1 – Fundamental and Integration Activities	
Work Package:	WP1 – SOA4All Runtime	
Due Date:		M18
Submission Date:		14/09/2009
Start Date of Project:		01/03/2008
Duration of Project:		36 Months
Organisation Responsible of Deliverable:		EBM WebSourcing
Revision:		1.0
Author(s):	Christophe Hamerling EBM Virginie Legrand INRIA Francoise Baude INRIA Elton Mathias INRIA Cristian Ruz INRIA Michael Fried UIBK Reto Krummenacher UIBK Philippe Merle INRIA Nicolas Dolet INRIA	

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**X**

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	2009-07-15	Initial TOC	Christophe Hamerling (EBM)
0.2	2009-08-01	Add PEtALS ESB section	Christophe Hamerling (EBM)
0.3	2009-08-11	Add section on message routing	Virginie Legrand (INRIA)
0.4	2009-08-24	Add monitoring platform section	Dong Liu (OU)
0.5	2009-08-24	Merge contributions. Update DSB section.	Christophe Hamerling (EBM)
0.6	2009-08-25	Update DSB installation and configuration sections	Christophe Hamerling (EBM)
0.7	2009-08-25	Introduction	Reto Krummenacher (UIBK)
0.8	2009-08-26	SCA Integration section	Christophe Hamerling (EBM), Philippe Merle (INRIA), Nicolas Dolet (INRIA)
0.9	2009-08-28	JBI Integration section	Christophe Hamerling (EBM)
1	2009-09-14	Final Editing	Malena Donato (ATOS)

Table of Contents

List of Figures

List of Tables

Glossary of Acronyms

Acronym	Definition
API	Application Programming Interface
CBSE	Component-Based Software Engineering
CDK	Component Development Kit
D	Deliverable
D	Deliverable
DSB	Distributed Service Bus
EC	European Commission
ESB	Enterprise Service Bus
JAX-WS	Java API for XML Web Services
JBI	Java Business Integration
SCA	Service Component Architecture
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
URL	Uniform Resource Locator
WP	Work Package
WP	Work Package
WS	Web Service
WSDL	Web Service Definition Language
XML	eXtended Markup Language

Executive summary

This report complements D1.4.1B, the first implementation of the SOA4All Distributed Service Bus, as core infrastructural service of the SOA4All Runtime. As such, this document is included in the zip file that contains the SOA4All Distributed Service Bus software, source code, and installation and configuration files.

The implementation presented in this deliverable yields the realization of the concepts and specifications that were published in deliverable D1.4.1A SOA4All Reference Architecture Specification [11] – mainly in Section 4.

1. Introduction

This report about the realization of the SOA4All Distributed Service Bus complements the first implementation of the SOA4All Runtime prototype. The prototype delivers the service bus logic implemented and deployable on top of the distributed ProActive Grid infrastructure. This work thus includes the implementation of the distributed technical registry of the bus, the messaging infrastructure for the coordination of distributed bus nodes, and the monitoring platform. A forth part that is deployed as part of the Distributed Service Bus is the semantic spaces infrastructure. The semantic spaces are not subject to this deliverable, but their implementation is documented under task T1.3 in deliverable D1.3.2B [12].

Further aspects that are subject to this deliverable are the various integration interfaces that are provided by the service bus infrastructure. In particular, we present details of how to publish services, in particular – for the time being – SOA4All Platform Services, to the runtime implementation. In continuation of this work, as part of the effort in the upcoming period of the project, the integration interfaces and guidelines that are published in this document, will be exploited for the realization of functional processes, as they were initially presented in deliverable D1.4.1A.

Services in SOA4All can in consequence be realized either as stand-along Web services based on traditional WS*-stack technologies or as RESTful service APIs, or however, with little more programmatic effort, they can be hosted by bus nodes in form of SCA or JBI components. The implementation guidelines and technical details about these approaches are subject to this deliverable too.

1.1 Purpose and Scope

The goal of this deliverable is to provide a written complement to the SOA4All Runtime prototype (D1.4.1B SOA4All Runtime). We refer the reader to D1.4.1A for details about technical background and the baseline, as well as the architecture and design of the prototype implementation. This deliverable is included as part of the zip file, D141B.zip, which contains the software, source code, installation and configuration facilities.

The main objective is thus to provide the reader with more detailed insights about the runtime implementation. Secondly, this deliverable offers more detailed insights into the technical integration of SOA4All Platform Services, or further third-party Web services. In that sense, this deliverable contains technical pointers towards the extended integration work of task T1.4 that will have to be done in the period up to month M24 of the project. By then, full-fledge functional processes and their implementation plans have to be specified. The purpose of these processes is to execute complex task in the SOA4All Runtime, such as the automated localization or composition tasks that require the coordinated execution of several platform services.

1.2 Structure of the document

In order to best respond to the purpose of the deliverable, we structure the document into the following sections. After this short introduction, Section 2 is dedicated to the publication of installation and configuration guidelines that allow the reader to run SOA4All Distributed Service Bus nodes. Section 3 presents the various bus APIs that are offered for easier integration of platform services (Section 3.1), or for accessing monitoring data (3.2). In Section 4 the various software components that constitute the bus are presented in more detail; namely, the distributed technical registry, the message transport infrastructure, and the monitoring platform. Before concluding with Section 6, we provide in Section 5 the guidelines and implementation details of how to register and expose services in the bus. This information is mainly dedicated to service developers, and as such, in particular to the implementers of SOA4All Platform Services.

2. Installation and Configuration

This section introduces how to install and configure pieces of software which are required to build the SOA4All Distributed Service Bus. The SOA4All DSB is mainly build by combining the OW2 PEtALS Enterprise Service Bus, the OW2 ProActive framework and the Space implementation described in D1.3.1.

The following section will describe all the different steps to follow to build what will be called the “SOA4All DSB node network”. The SOA4All DSB node network is the virtual network which is composed of SOA4All Service Bus nodes. These nodes are composed of the PEtALS, ProActive and Space runtimes and have to be configured to provide a static node network. By static, it means that nodes can not discover other nodes over the Internet automatically. This implies to define a network topology, to ensure that all the required services will be publically exposed and then install and configure the nodes according to this topology.

2.1 Resources Requirements

In order to provide the SOA4All Distributed Service Bus platform, the following resources are required to install a node:

- Host with:
    - 10 Gb free disk space (recommended)
    - 1 Gb RAM (recommended since PEtALS ESB needs 512 Mb to work properly)
    - Java Runtime Environment (SUN MicroSystems JRE 5 at least)
    - Full Internet access: This point is the most important since the DSB will be used to consume and provide billions of services over the Internet.

A DSB node needs to expose services (Web services over HTTP) to other DSB nodes. The list of ports which must be open is:

- 8084: Port used by the PEtALS SOAP Binding Component to expose services
- 7100: Port used by the PEtALS monitoring and management Web services
- 7800: Port used by the node communication layer
- 7900: Port used by the distributed technical registry

2.2 Infrastructure

The SOA4All DSB is composed of software modules which have been extended and integrated to provide a solid, scalable and extensible infrastructure.

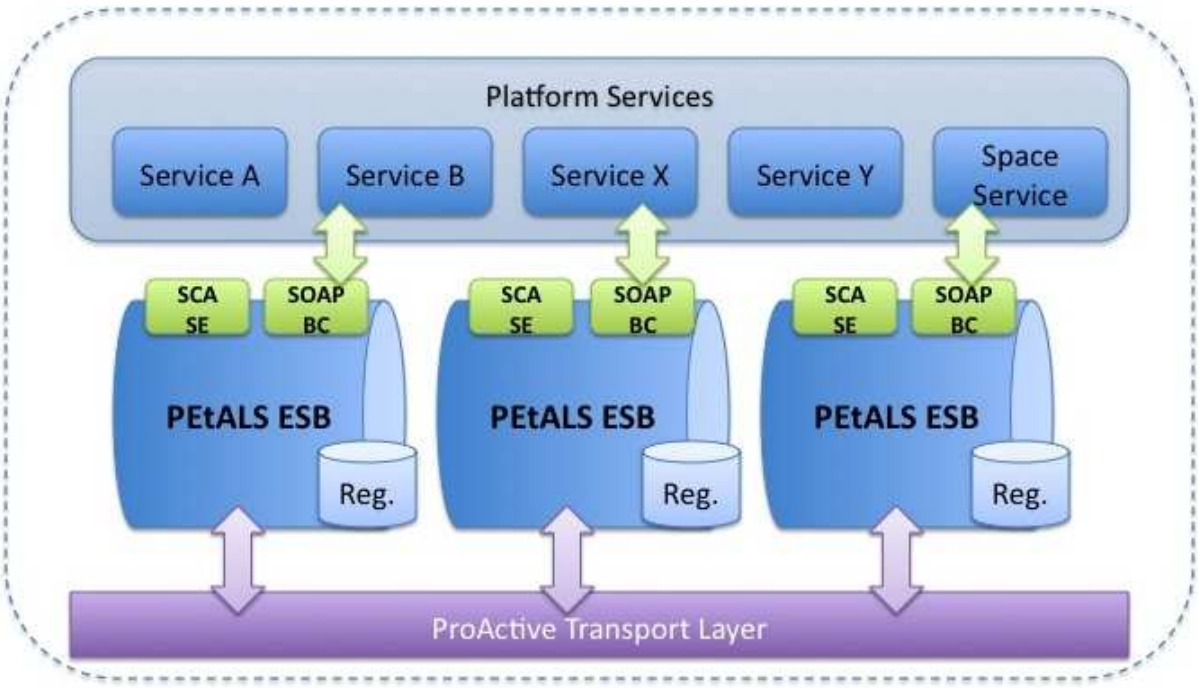


Figure 1 SOA4All DSB Infrastructure

The previous figure describes the SOA4All DSB infrastructure and shows how modules are integrated together.

- The PEtALS ESB has been extended to provide:
  - Additional services such as platform services for service binding, monitoring and management services
  - New scalable and firewall-friendly technical registry
  - New message transport layer based on ProActive

  A complete description of the standard OW2 PEtALS ESB architecture is available at (http://petals.ow2.org/docs/PEtALS-Architecture-3-07-09.pdf).
- The ProActive framework is used to build the message routing and transport layer. This layer provides the communication feature needed by the PEtALS ESB.
- The platform services are potentially hosted on external nodes as Web services. These services are binded to the DSB using the PEtALS SOAP Binding Component. Such bound platform service is accessible from each DSB node in a transparent way.
- The platform services can be composed using the SCA capabilities with the help of the PEtALS SCA Service Engine. This is only possible if the platform services are bound to the DSB using the service binding feature.
- The Space is hosted on a separate runtime providing a Web service API. The Space Web service is bind to the DSB using the PEtALS SOAP Binding Component.

2.2.1 Installating a DSB node

The DSB binary is available as a ZIP archive. To install the DSB node, just extract the archive. The DSB runtime is then available under *SOA4All-dsb* folder and will be ready to be launched after the following configuration phase.

2.2.2 Configuring a DSB node

The SOA4All DSB distribution needs to be configured to fit the targeted system. Configuring the node means defining the network ports to be used and defining the SOA4All network topology.

The DSB configuration files are available under the *SOA4All-dsb/conf* folder :

- *server.properties* : Defines the local server properties such as name in the network, services to be exposed … These informations are locally and not shared with others nodes.
- *topology.xml* : Defines the local network topology. The local network is the set of DSB nodes which are visible from the local one. The topology information is shared between nodes of the same local network so that all gets the same vision of the network.

2.2.2.1 Local node configuration

The local configuration is defined under the *conf* folder in the *server.properties* file. This file is a standard properties file with '*key=value'* data. The values to be updated according to the DSB configuration are:

- The node name '*petals.container.name=X*' where X is the name of the node in the SOA4All DSB node network. This name MUST be unique and must match a node entry in the topology definition.
- The registry definition:
  - *'registry.incoming.manager=org.ow2.petals.registry.core.strategy.flooding.SequentialFloodingIncomingManager'* defines the class name to use for the incoming message propagation from other registry instances. The class must be available in the classloader and must implements *org.ow2.petals.registry.api.manager.IncomingManager*.
  - *'registry.outgoing.manager=org.ow2.petals.registry.core.strategy.flooding.SequentialFloodingOutgoingManager'* defines the class name to use for outgoing message propagation. The class must be available in the classloader and implements *org.ow2.petals.registry.api.manager.OutgoingManager*.
  - *'registry.message.receiver=org.ow2.petals.registry.core.transport.cxf.CXFMessageReceiver'* defines the class name to use for message reception from other registries instances. The class must be available in the class loader and must implements *org.ow2.petals.registry.api.transport.MessageReceiver*.
  - *'registry.message.sender=org.ow2.petals.registry.core.transport.cxf.CXFMessageSender'* defines the class name to use for message emission to other registries instances. The class must be available in the class loader and must implements *org.ow2.petals.registry.api.transport.MessageSender*.

2.2.2.2 Topology definition

The topology configuration is defined under the *conf* folder in the *topology.xml* file. This file contains all the nodes configuration data such as host names and port numbers. The following configuration snippet shows a topology definition for three DSB nodes:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:topology xmlns:tns="http://petals.ow2.org/topology"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://petals.ow2.org/topology petalsTopology.xsd">
```

```
      <tns:domain mode="static" name="SOA4AllDSB">
            <tns:description>The SOA4All DSB domain configuration</tns:description>
            <tns:sub-domain name="subdomain1">
                  <tns:description>description of the subdomain</tns:description>
                  <tns:container name="ebmws">
                        <tns:description>description of the container hosted at eBM WebSourcing</tns:description>
                        <tns:host>SOA4All.ebmwebsourcing.com</tns:host>
                        <tns:user>petals</tns:user>
                        <tns:password>petals</tns:password>
                        <tns:network-service>
                              <tns:port>7720</tns:port>
                        </tns:network-service>
                        <tns:jmx-service>
                              <tns:rmi-port>7700</tns:rmi-port>
                        </tns:jmx-service>
                        <tns:transport-service>
                              <tns:tcp-port>7800</tns:tcp-port>
                        </tns:transport-service>
                        <tns:registry-service>
                              <tns:port>7900</tns:port>
                        </tns:registry-service>
                  </tns:container>
                  <tns:container name="inria">
                        <tns:description>description of the container hosted at INRIA</tns:description>
                        <tns:host>SOA4All.inria.fr</tns:host>
                        <tns:user>petals</tns:user>
                        <tns:password>petals</tns:password>
                        <tns:network-service>
                              <tns:port>7720</tns:port>
                        </tns:network-service>
                        <tns:jmx-service>
                              <tns:rmi-port>7700</tns:rmi-port>
                        </tns:jmx-service>
                        <tns:transport-service>
                              <tns:tcp-port>7800</tns:tcp-port>
                        </tns:transport-service>
                        <tns:registry-service>
                              <tns:port>7900</tns:port>
                        </tns:registry-service>
                  </tns:container>
```

```
                    <tns:container name="sti2">
                        <tns:description>description of the container hosted at
STI2</tns:description>
                        <tns:host>SOA4All.sti2.at</tns:host>
                        <tns:user>petals</tns:user>
                        <tns:password>petals</tns:password>
                        <tns:network-service>
                            <tns:port>7720</tns:port>
                        </tns:network-service>
                        <tns:jmx-service>
                            <tns:rmi-port>7700</tns:rmi-port>
                        </tns:jmx-service>
                        <tns:transport-service>
                            <tns:tcp-port>7800</tns:tcp-port>
                        </tns:transport-service>
                        <tns:registry-service>
                            <tns:port>7900</tns:port>
                        </tns:registry-service>
                    </tns:container>
                </tns:sub-domain>
        </tns:domain>
</tns:topology>
```

In the previous configuration snippet, three nodes (*ebmws*, *sti2* and *inria*) have been defined. It is very important that:

- The local configuration match with the container name. For example, the DSB node instance running at eBM WebSourcing is named ebmws in the topology and must define *petals.container.name* to *ebmws* in the *server.properties* file.
- The topology file is the same on all nodes since this is the only way to share this information.

2.2.3 Starting and stopping a DSB node

Once configured, the DSB node can be managed with the help of scripts available under the *SOA4All-dsb/bin* folder:

- *startup.sh*: Starts the local DSB node. This will launch the local runtime and connects it to the SOA4All DSB network.
- *stop.sh*: Stops the local DSB node. This will stop the local runtime and remove it from the SOA4All DSB network.

2.2.4 Modules installation

The DSB needs to:

- Bind platform services using the PEtALS SOAP Binding Component
- Compose platform services using the PEtALS SCA Service Engine

To install these components, copy the *petals-bc-soap.zip* and *petals-se-sca.zip* files located in the *SOA4All-dsb/components* folder into the *SOA4All-dsb/install* folder.

The procedure to bind and compose services is explained in the section 5 DSB Integration.

3. SOA4All Distributed Service Bus APIs

This chapter describes the actual SOA4All Distributed Service Bus APIs. These APIs will be used by:

- Platform services consumers: The platform services will be exposed and potentially consumed or composed with other services. The SOA4All studio is the main platform service consumer. All the requests to platform services from the Studio to platform services will be performed through the SOA4All DSB API.
- Monitoring agents: The monitoring data will be used by monitoring agents to provide a higher level of monitoring. The SOA4All Studio is the main monitoring service consumer and will provide an advanced GUI representation of this monitoring data.
- Service managers: The service manager role is to bind existing services, which can be hosted on several places, to the bus. Once bound, these services are transparently accessible to all the platform service consumers.

3.1 Platform services integration API

The platform services integration API is described in DSB Integration.

3.2 Monitoring and Management API

3.2.1 Infrastructure Monitoring Data

The infrastructure layer (PEtALS ESB) provides a basic monitoring Web service API which allows clients to get standard information on hosted services such as:

- Messages exchanged between consumers and providers: with message payload, timestamps, states, …)
- List of available endpoints: with location, service description (WSDL, WADL, WSMO, …), number of requests,…
- List of PEtALS components: with location, state.

3.2.2 Monitoring Bus

The Monitoring Bus (MB) is an upper level bus which is placed on top of monitoring aware pieces of software. This MB acts as an intermediate layer between the infrastructure which produces monitoring data and clients which will consume or be notified of WSDM data availability.

Using the MB is possible with the following steps:

- The MB provides an API to create Monitoring Endpoints (ME). These endpoints are generally created by functional endpoints hosted on the monitored infrastructure.
- Data is sent from the monitored infrastructure to one or more ME.
- Clients use the MB with the subscribe/notify paradigm. A client subscribes to one or more ME and is notified when new monitoring data is available.

The generic WSDM monitoring interface is composed of the following operations:

- *GetResourceProperty*: Get all supported topics by the producer service.
- *Subscribe*: Allows a client to subscribe to one topic of provider.
- *Unsubscribe*: Allows a client to unsubscribe to one topic of provider.

- *GetCurrentMessage*: Allows a client to get the current message of a topic given in parameters.

3.2.3 Monitoring Service Integration

The next figure introduces the interaction between the WSDM based Monitoring Service (MS) and the infrastructure (PEALS ESB). The MS is deployed on the Monitoring Bus. This specific bus contains a Monitoring Admin Service which provides the capability to create or destroy a monitoring service endpoint on the MB.

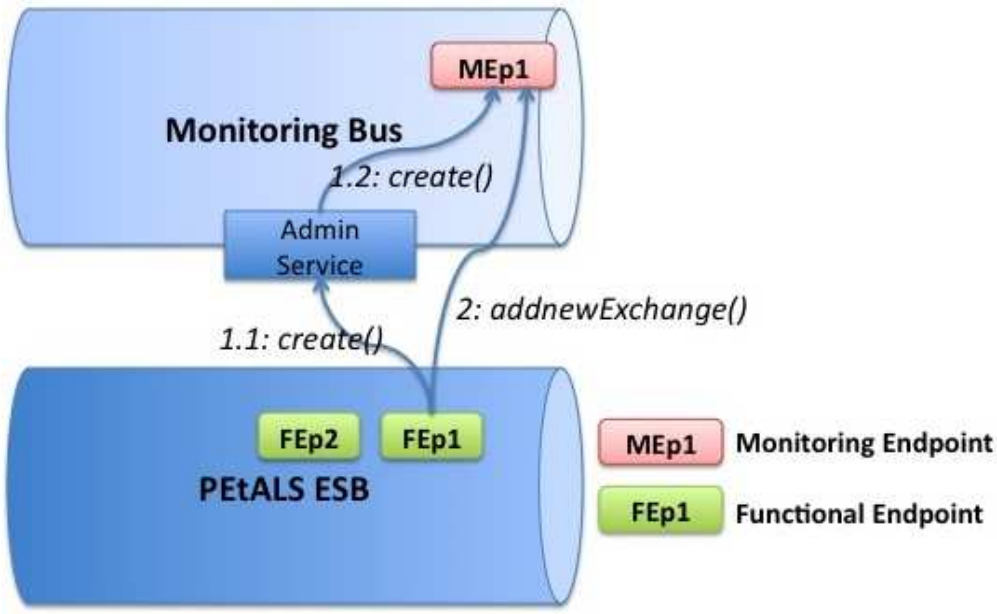


Figure 2 Connection between the Monitoring Bus and PEtALS ESB

When a functional endpoint is deployed on the PEtALS ESB, the requests the Monitoring Admin Service to create its associated monitoring endpoint in the Monitoring Bus. The monitoring endpoint address is returned as response of the create operation.

As a result, when the functional endpoint receives a client request, it processes the request and sends a snapshot of the exchange to the associated monitoring endpoint using the '*addNewExchange*' operation.

All information concerning the exchange between the functional consumer and provider are sent to the monitoring endpoint. This operation is contained in a private interface accessible only by the functional endpoint.

4. SOA4All Distributed Service Bus Components

The SOA4All Distributed Service Bus is composed of various pieces of software. The main ones are:

- The OW2 PEtALS Entreprise Service Bus
- The OW2 ProActive middleware framework

As described in the D1.4.1A deliverable, these pieces of software must be adapted to provide features such as better scalability, proxy compatibility, efficient message transport and message routing. The role of the current chapter is to introduce the specific modules which have been developed to match the previous requirements.

4.1 Distributed Technical Registry

4.1.1 Requirements

The technical registry is the software module where Service Bus service endpoints are managed. The registry API follows the CRUD paradigm to create, retrieve, update and delete endpoints.

In order to access services, the Service Bus resolves endpoint location by doing a lookup to the technical registry. The technical registry must provide the following core features:

- Store and retrieve entries into/from the registry. An entry is composed by the endpoint name, the endpoint location (DSB node level), the service description (WSDL, WADL, WSMO i.e. XML description).
- Store and retrieve entries from any node of the SOA4All Network. This means that a DSB node must be able to get a reference to an endpoint which is hosted by another DSB node. This will enable the endpoint retrieval when a DSB node hosting the endpoint is down for some reason. The DSB based on PEtALS ESB will be able to send the message to the right service when the node will come back.
- The registry entries propagation must be configurable. The DSB network will be potentially composed of hundreds of nodes of different natures over different physical networks through proxies and/or firewalls. Propagating and getting entries through all these nodes in order to have the current vision of the available services will be configurable to get the most effective result.
- Entries must be persisted. Entries will be persisted to be able to restart the technical registry without restarting the complete DSB node.
- The transport layer will be configurable. The messages between registries will be exchanged by using a transport layer. Since DSB nodes may be accessible through firewall and/or proxies, the transport layer may be configurable to use the right protocol to access foreign nodes.

4.1.2 Architecture

The technical registry architecture resulting of the previously features and requirements described above is represented in the following figure.

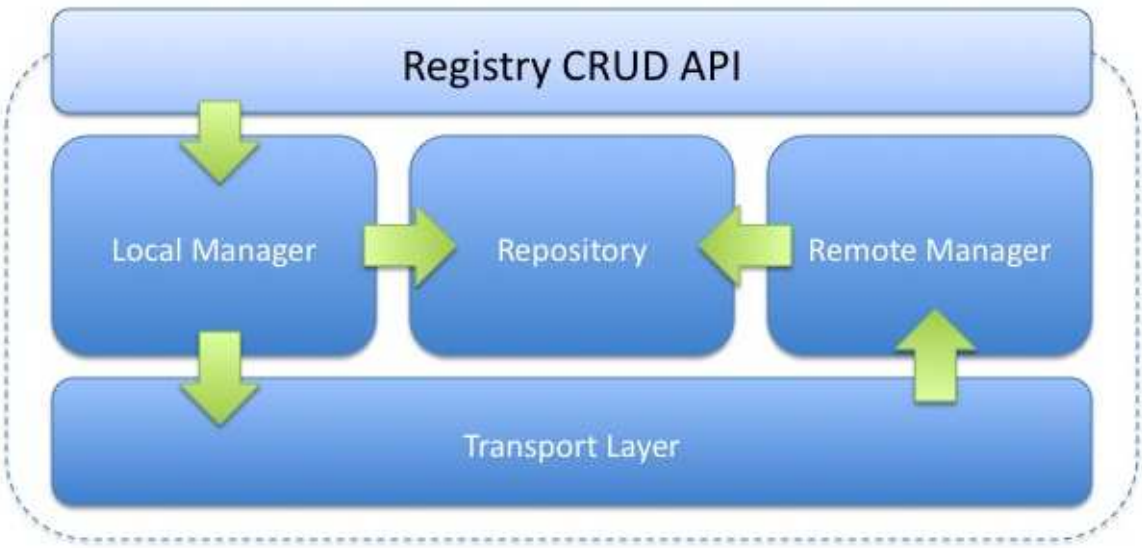


Figure 3 Technical Registry Architecture

The registry architecture is composed of the following modules:

- *Registry*: The registry module is providing the registry API to clients. It is used to store and retrieve entries.
- *Repository*: The repository module is used to store entries.
- *Transport*: The transport layer is used to send and receive messages between registries. The messages can content entries or management data.
- *Remote Manager*: The remote manager defines how the entries must be propagated to known peers.
- *Local Manager*: The local manager defines how the entries must be handled when they are received from remote peers i.e. the entry must be propagated to other peers, how, when?

As the registry interface is generic and modularized, the transport layer can be customized. It can be for example, based on the message transport described in section 4.2. We believe that such a way to propagate requests is relevant w.r.t. the DSB topology. The requests are propagated through the distributed message transporter (which boundaries are well-defined).

4.2 Message transport

4.2.1 Overview and architecture requierements

According to the SOA4All vision, the DSB infrastructure, which can be seen as a federation of ESB, should cope with billions of external services, orchestration enactment of thousands of compound services involving subsets of those billions above;millions of users concurrently accessing and deploying services, relying on the bus in the background, through the web-enabled so-called SOA4All studio, a few thousands of them acting as compound service designers.The primary goal of the SOA4All DSB is to ensure that the SOA4All framework scales to the dimensions of the Web, by enabling appropriate distribution techniques that evolve the traditional ESB towards a fully Distributed Service Bus, without altering the communication and interaction patterns of the ESB core.

As a consequence, an ESB by itself, even if distributed, is not a suitable solution for a service cloud resulting of the federation of the involved partners service infrastructures, e.g., their service buses. It mainly lacks inter-connection mechanisms and a global and shared store of meta-data about federated services. Additionally, each infrastructure should be able to dynamically add or remove additional nodes to face changing loads. Such a demand may also translate at the federation level where, e.g. some agreements may have been set up to govern the load-balancing of underlying hardware or software resources among the federated buses.
The most crucial element to build the PEtALS-based DSBs is to provide a solution for message routing through a potentially huge set of PEtALS nodes, without posing a constraint of having point-to-point connections among all the containers, and without using the original JMS broker (based on Joram, see **Error! Reference source not found.**) which would be difficult to deploy at Web scale. Naturally, we came out with a multi-level, hierarchical organisation, presented in section 4.2.3.

4.2.2 Message transport in PEtALS

In a non-distributed version, the PEtALS message transport relies on a so-called Message Transporter Fractal component (which is a Julia component).

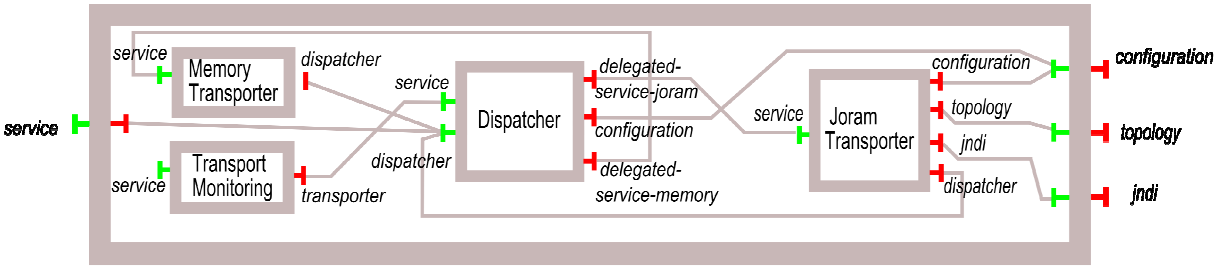


Figure 4 The Transporter Component

The transporter component, shown in **Error! Reference source not found.**, handles the transport of messages to other buses. It's accessed through the Dispatcher that transmits messages to the right transporter. When a message arrives, it's sent to the appropriate TransportProtocol, which may be:

- MemoryTransporter, in case of a local container.
- JoramTransporter, for "reliable transporting"
- Another kind and customizable transporter,

4.2.3 The distributed message transporter

Our proposition relies in a new transport component of a PEtALS DSB including a new transport layer based on GCM components. Each DSB includes a GCMTransporter component which is responsible for the transparent support to point-to-point and collective communications over the Grid/Cloud infrastructure. PEtALS containers *message transporters* subcontract the transport to a GCM/ProActive-based hierarchical grid-aware routing platform [1]. This platform leverages the hierarchical Fractal composition model: activities deployed on a same cluster of machines can use standard communication protocols (RMI over TCP) to exchange messages; on the contrary, the messages can be tunneled across SSH or any other technology supported by the ProActive middleware.

To this effect, the GCMTransporters are organized hierarchically at deployment-time according to the multi-cluster/grid/cloud physical infrastructure, which compose the federation of DSBs. **Error! Reference source not found.**. shows an example of the routing platform composition on an infrastructure composed by two distinct resource providers, containing respectively 4 and 'n' ESBs.

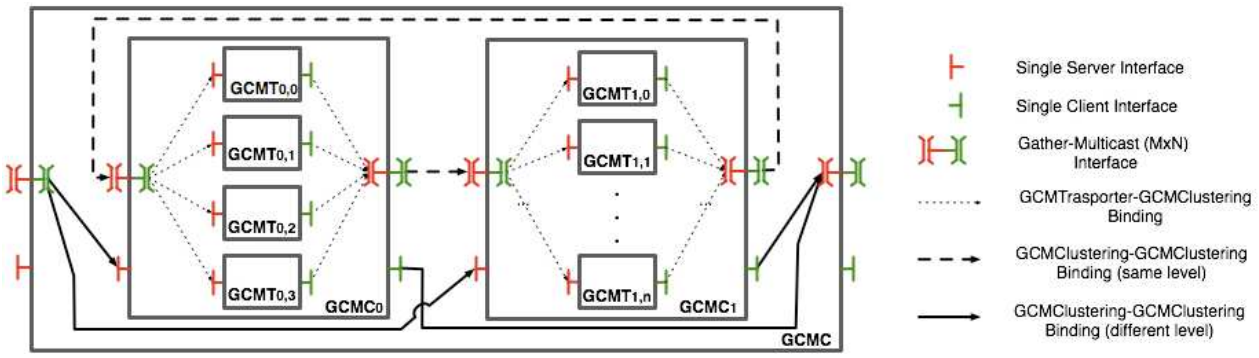


Figure 5: Example of DSB routing platform composition

This Platform is composed by two main kinds of components:

- GCMTransporters (GCMT): these are primitive GCM components (inner-components in **Error! Reference source not found.**), which are responsible, locally, for message handling, that basically consist on adding meta-information, which will guide the routing strategy. These components have a hierarchical identifier (ID) composed by the ID of the enclosing GCMClustering plus an unique ID. In one hand, communications that place in the context of a cluster (or local network) will be handled by the GCMTransports themselves and such communications are usually performed through the standard RMI protocol. On the other hand, messages exchanged among GCMTransporters located on different administrative domains are delegated to GCMClustering components. **Error! Reference source not found.** shows internally, how the GCMTrasporters are organized.

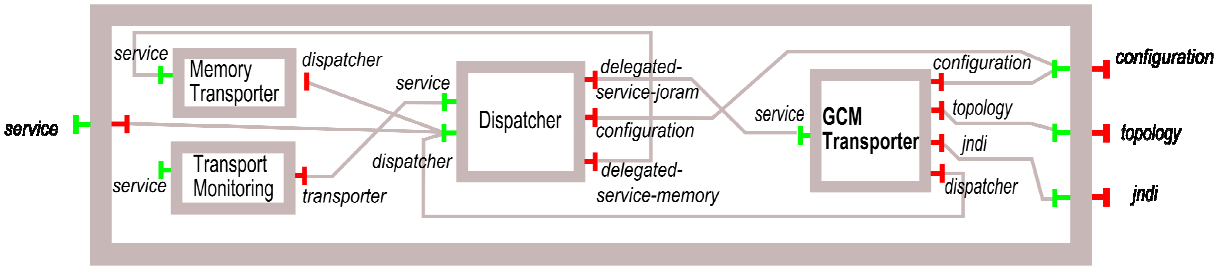


Figure 6: The transporter component based on GCM

- GCMClusterings (GCMC): These are composite GCM components (enclosing components of **Error! Reference source not found.**), which are capable of grouping a set of GCMTransporters or a set of lower-level GCMClusterings. These components also have a hierarchical ID, which is composed by the ID of the enclosing GCMClustering, if they have any, and a

single ID. The GCMClusterings are responsible for message routing, which is achieved through the configuration of their collective MxN interfaces.

The GCMCs are deployed on resource provider frontends or proxies to make possible the communication among physically isolated nodes. This can be achieved through the usage of specific communication protocols, such as RMISSH (RMI over SSH tunnels).

The binding scheme of the infrastructure depicted on **Error! Reference source not found.** consists on single bindings between GCMTs and the respective enclosing GCMCs, all-to-all bindings among GCMCs of the same level (even if in our example e only have 2 GCMCs) and single bindings between GCMCs to upper-level GCMCs. This is a very general scheme that allows point-to-point and collective communication from any GCMT or group of GCMTs to any other group of GCMTs, even vith limited network connectivity, provided a physical link between the most external GCMCs. Externally, the whole infrastructure is wrapped bya GCMC that represents the whole infrastructure and can potentially be bound to another routing infrastructure.

In practice, messages sent from GCMTs are tagged with:

- Message type: point-to-point or collective (1xN, Mx1 or MxN).
- Destination (s): a given hierarchical ID, in the case of point-to-point or Mx1 or a set of hierarchical IDs, in the case of 1xN (which can potentially be a broadcast if N is composed by every other destination) and MxN.
- And distribution policy which are functions that will be used to split or gather the messages.

As already explained, if messages take place on the local network, they are sent directly to the destination. Otherwise, these messages are intercepted at the gather-multicast and a routing algorithm allows the messages to be delivered to the right destination(s), applying the distribution policy, if needed.

The routing infrastructure provides a very general mechanism capable of performing any kind of communication, supposing one open channel between clustering components of the same level (even if it must be tunnelled through SSH or use HTTP). However, in terms of performance, this cannot be considered an optimal solution by any means: one or two synchronization barriers for each level involved on the call are required while efficient point-to-point operations require some form of direct communication between source and destination. Besides, the memory needed to aggregate messages at interfaces might be considerable. Nonetheless, collective can take profit of the gather-multicast interface to stage and parallelize their execution.

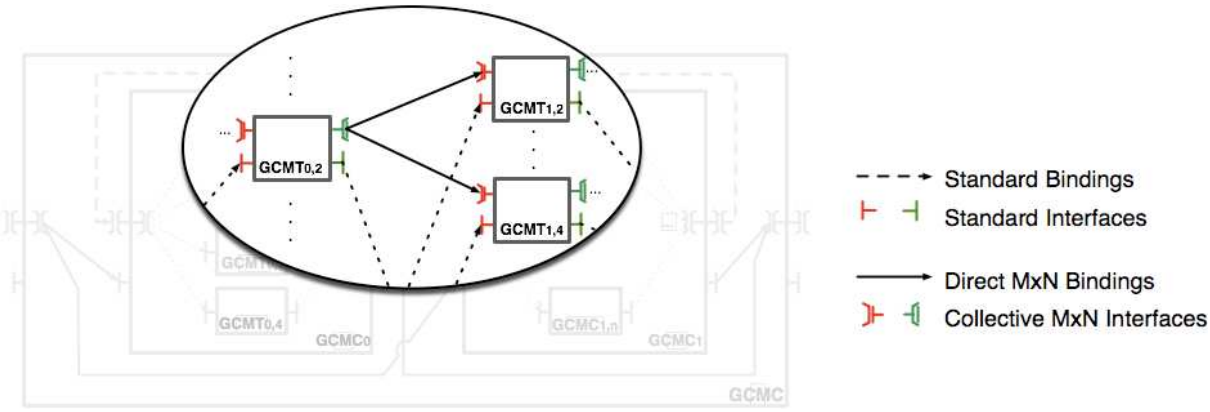


Figure 7 : Routing optimization through direct bindings

Thanks to the component encapsulation, we are able to transparently configure direct bindings (**Error! Reference source not found.**) among GCMTransporters to avoid unnecessary indirections, provided an existing physical link between each pair of nodes. The general direct binding mechanism is detailed in [2].

4.3 Monitoring platform

Error! Reference source not found. shows the overall architecture of the monitoring platform. The monitored components running on the DSB, e.g. the discovery engine, execution engine and reasoning engine, etc. continually place monitoring data onto the message queues on the DSB, which is in the form of OWL encoded monitoring events. Each event indicates a transition of state of execution, for example, the event *ActivityCompleted* means that an activity has been finished successfully. On the other hand, the monitoring platform sets up several listeners to the message queues in order to continuously wait for monitoring events to occur. The listeners can see the events happening, and pass them to the monitoring platform for processing. When events arrive, the monitoring platform firstly stores them as RDF statements in the Event Repository, and secondly puts the high-level information derived through rule based reasoning on the raw monitoring events, into the Execution History.

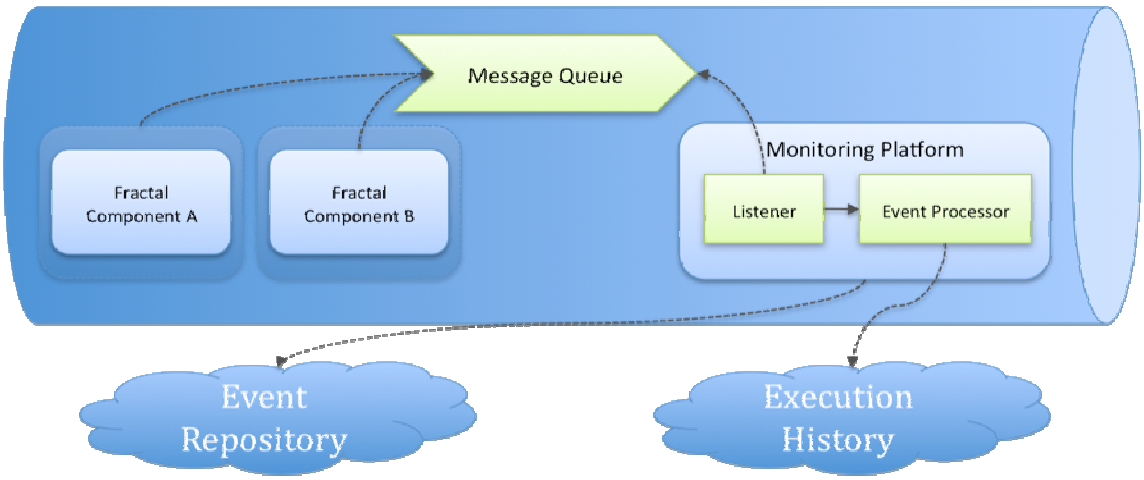


Figure 8 Architecture Monitoring platform

4.3.1 Underlying infrastructure

The monitoring platform is situated in the context of the SOA4All Distributed Service Bus, and hence the monitoring data is collected through the monitoring service provided by the PEtALS ESB. As stated in the previous section, information provided by monitoring services on ESB not only can capture the messages exchanged during services invocation and execution of business processes, but also can reflect the runtime states of message queue and JBI artefacts lifecycles.

Message queues on the DSB provide an asynchronous communication mechanism to transport monitoring data to the monitoring platform. Furthermore, with the supports of WS-Topics [11], administrators or analysts can apply specific filters to the events that are generated by the components on DSB, and thereby screen out information that is relevant to the monitored artefacts they concern. By this means, the communication loads of the message queues are reduced, and the general performance of the monitoring platform is improved. In addition, users of the monitoring platform can store the gathered data into separate repositories, namely publish them to different sub-spaces of the Semantic Spaces [9]. For instance, as shown in **Error! Reference source not found.**, the system administrators need to see the whole picture of the running state of the DSB, and receive all the monitoring data by using the root topic. In contrast, the business analysts are only interested in the special activities, so the corresponding topics, such as '*super.events.ode*', are adopted, as well as the semantic repositories with relative limited size. In brief, the underlying infrastructure makes it possible to observer the SOA4All DSB from multiple perspectives, and to obtain the required data through the monitoring platform.

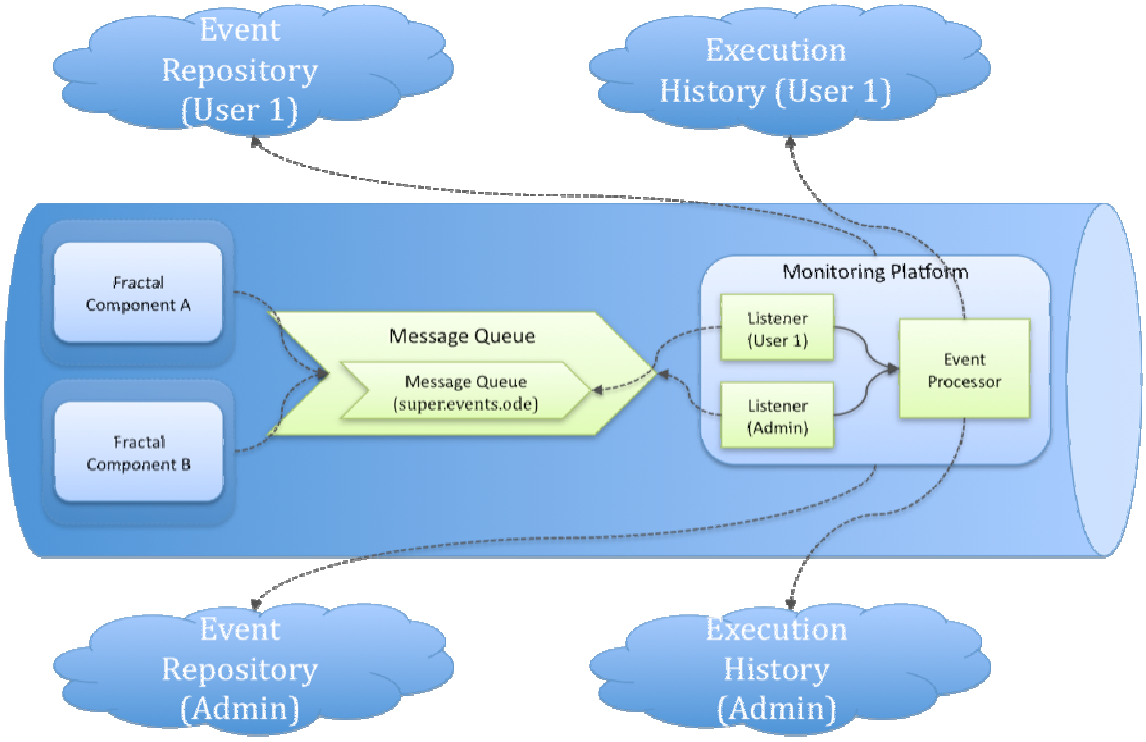


Figure 9 Topic-based Monitoring Data Filtering

4.3.2 Persistence of monitoring data

The persistence of monitoring data refers to writing runtime information to RDF triple storage systems, which includes both the gathered raw events generated by components of SOA4All DSB and the high-level information derived through event processing. In this way, the system administrators and business analysts can keep track of the runtime states and execution

histories with much longer time-span.

From an architectural viewpoint, as shown in **Error! Reference source not found.**, the storage system of monitoring data is composed of two separate semantic repositories: Event Repository and Execution History Repository. The former one is assigned for the raw events that depict the source, the occurrence time, the related input and output data, etc. The Execution History Repository is used to store the high-level information, i.e. instances of the concepts in the COBRA ontology [6]. Once raw events arrive the monitoring platform, they are resolved as instances of the concepts in the EVO ontology [6], and after being processed by the rule-based reasoning, the derived high-level information is published to the Execution History.

The reasons for creating two repositories for respectively storing the raw events and high-level information are:

1. A great number of events may happen in a short time, so a dedicated storage can reduce the impact on the overall performance of the monitoring platform;
2. business analysis is mainly based on the high-level information, e.g. the lifecycles of business activities, rather than the raw events, thus it does not need the access to all of the repositories. Moreover, the design of two separate semantic repositories is also for the purpose of decoupling the event processing of monitoring and other functionalities of business analysis.

The persistence of monitoring data is implemented under the framework of Elmo, which facilitates Java applications accessing RDF triple stores by establishing the mappings between Java objects and RDF resources [7].

4.3.3 Event processing

The event-processing module of the monitoring platform is charge of deriving the high-level information from the raw events and publishing to the Execution History Repository. Drools [8], a general-purpose rule engine, is exploited to carry out event processing. Since we have introduced a hierarchy of monitoring events in the EVO ontology (refer to [12] for more details about EVO ontology). When an event arrives the monitoring platform, SPARQL [9] queries are executed to retrieve the super-concepts of the received event. Then, the event itself and its super-concepts are translated into Drools facts and fed to the rule engine. To achieve the automation of data updating in Execution History Repository, reasoning rules are defined for each kind of event. A new feature of Drools 5.0 that is useful for event processing is the support to '*fireUntilHalt()*', which means the engine can run in a reactive mode and fire the rules until the function '*halt()*' is called.

The monitoring data is kept in the working memory of the rule engine for only a certain period of time. In other words, only the events that happened during a specific period of time are treated as the facts on which rule engine reasons. By this means, a sliding window of observation on events comes into being, and the number of monitoring events being processed by the rule engine is controllable. Therefore, the processing time is cut down and the system performance is also improved.

To integrate Drools with Elmo, two approaches are adopted. For one thing, the Java class loader used by Elmo is passed on to Drools, so that the rule engine can deal with Java objects that are instantiated by Elmo. For another, while constructing the specification of the reasoning rules, the Java interfaces generated by Elmo is imported. Furthermore, the concepts defined in the EVO ontology are declared as events that can be detected by Drools. Finally, the results of rule-based reasoning are persisted through the interface to the RDF triple store, which is provided by Elmo.

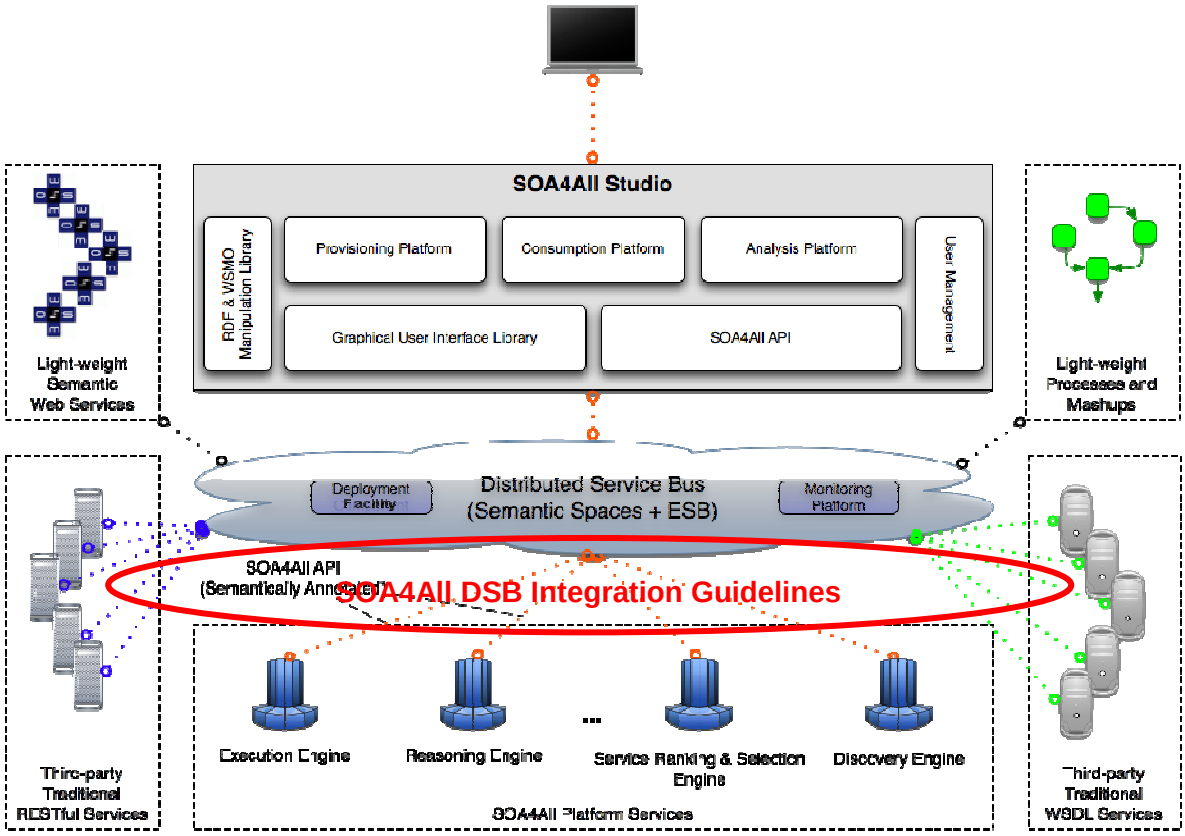
5. DSB Integration

5.1 SOA4All Architecture and Integration Guidelines

This chapter reminds the SOA4All overall architecture described into D1.4.1A [11], provides an overview of SOA4All DSB integration guidelines, and discusses the pros and cons of the three complementary approaches to plug SOA4All Platform Services to the SOA4All DSB.

5.1.1 SOA4All Overall Architecture

The SOA4All overall platform is composed of the SOA4All Distributed Service Bus (DSB), the SOA4All Studio, and several SOA4All Platform Services as depicted by **Error! Reference source not found.** and **Error! Reference source not found.** extracted from



D1.4.1A [11][1].

Figure 10: The SOA4All Overall Architecture

The SOA4All DSB will be hosted on a set of distributed machines, which are publically accessible through Internet and at first provided by EBM WebSourcing, INRIA and STI. These machines are also called SOA4All DSB nodes.

As shown by **Error! Reference source not found.**, the SOA4All DSB is the backbone transporting all the communications/interactions from the SOA4All Studio to Platform Services or external business services, from SOA4All Platform Services to external business

[1] For more details, read Chapter 2 of D1.4.1A [11].

services, and between SOA4All Platform Services.

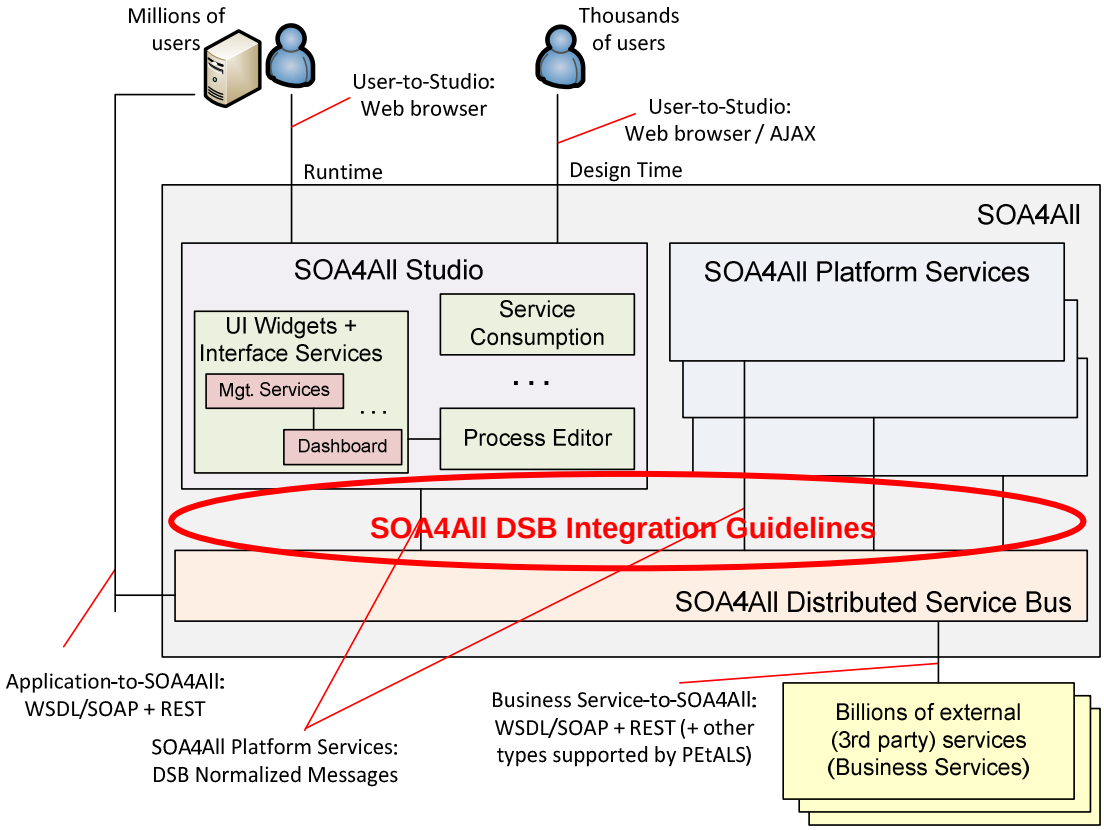


Figure 11: Internal and External Communication Flow

In **Error! Reference source not found.** and **Error! Reference source not found.**, the red circle points out that SOA4All DSB Integration Guidelines apply at the interface between the SOA4All DSB and Platform Services. These guidelines define how SOA4All Platform Services can be plugged to the SOA4All DSB.

5.1.2 Overview of SOA4All DSB Integration Guidelines

The SOA4All DSB Integration Guidelines define three approaches to plug SOA4All Platform Services to the SOA4All DSB, called standalone, SCA-based, and JBI-based respectively, and depicted into [2].

[2] For more details about this figure, read Chapter 8 of D1.4.1A [11].

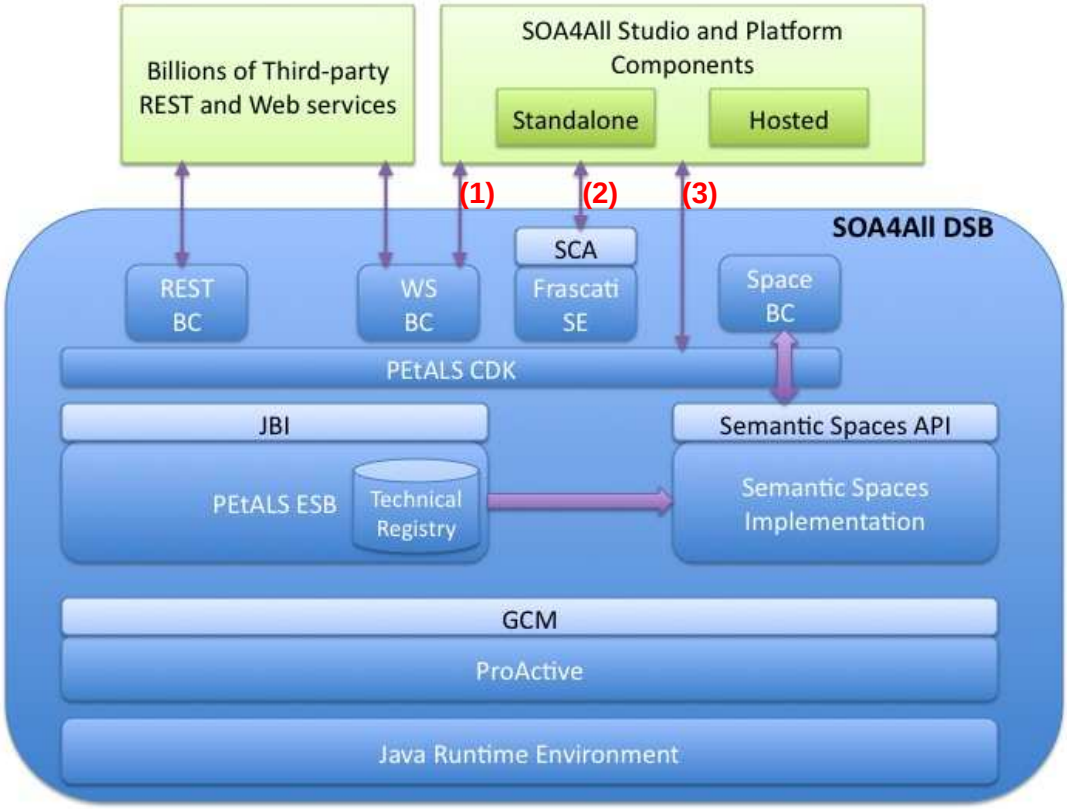


Figure 12: The SOA4All DSB Implementation Architecture

These three integration approaches are not exclusive but complementary. Developers of each SOA4All Platform Service can choice one of these approaches according to pros and cons discussed into next sections.

5.1.2.1 The Standalone Approach

A platform service can be a public external Web Service. It runs as a standalone program but is not hosted by one of the SOA4All DSB nodes. The platform service must be exposed to the SOA4All DSB in order to be accessible by other SOA4All components and accesses other platform services via the Web Service Binding Component (WS BC) provided by the SOA4All DSB. This is the responsibility of its developers to host this platform service on their own public machines.

5.1.2.1.1 Pros

- **Traditional Web Service development process**. The main advantage of the standalone approach is that SOA4All developers can use any traditional Web Service development process and their favorite tools to implement their platform services, e.g., defining the platform service interface by a WSDL document, using a *wsdl2java* tool to generate Java interfaces and classes conform to the Java API for XML Web Services (JAX-WS) [13], implementing their business code by inheriting from and calling JAX-WS-based generated interfaces and classes. Developers could also start from Java business interfaces annotated with JAX-WS and use a *java2wsdl* tool to generate the corresponding WSDL document.

5.1.2.1.2 Cons

- **No hosting facility provided**. The main disadvantage of the standalone approach is

that SOA4All does not provide public machines to host these standalone platform services. This is the responsibility of its developers to host their platform services on their own public machines. They must provide and administrate their own public machines, they must deploy and manage their SOA4All Platform Services, they must manage themselves hardware/software faults and availability, and they must do their best efforts to make their platform services always accessible, even after the end of the SOA4All project.

- **No DSB extra features**. As standalone platform services are not hosted by the SOA4All DSB then they can not benefit from its advanced extra features like distribution, automated deployment, lifecycle management, monitoring, etc.
- **No optimal communication performance**. As standalone platform services are hosted outside the SOA4All DSB and are only accessible as Web Services (e.g., SOAP over HTTP), the SOA4All DSB has no opportunity to optimize communications between platform services, i.e., it will always use WS bindings. Then the performance of communications (i.e., latency, throughput) would certainly be not optimal.
- **No assistance for better reusable software components**. Traditional Web Services development process (e.g., with WSDL and JAX-WS) helps a lot to make a piece of software accessible as a Web Service but does not assist developers to develop highly reusable software components. Component-Based Software Engineering (CBSE) provides guidelines and technologies (e.g., EJB, Spring, OSGi, JBI, and SCA) to produce better reusable software components by promoting a clear separation between the code dealing with business, configuration, and non-functional concerns of applications. Business code is implemented with a programming language, configuration is described with a dedicated language (e.g., XML-based descriptors), and non-functional concerns are managed by component containers transparently.

5.1.2.2 The SCA-based Approach

A platform service can be implemented as an SCA-based composite application[3]. Then this platform service can be fully hosted, deployed, and monitored, at first, on the SOA4All DSB node provided by EBM WebSourcing.

5.1.2.2.1 Pros

- **Hosting facility provided**. At first, EBM WebSourcing will provide a SOA4All DSB node to host SCA-based platform services. Then SOA4All developers need not to provide and administrate their own public machines. The SOA4All DSB provides tools to facilitate the deployment, management, and monitoring of SCA-based platform services.
- **DSB extra features**. As SCA-based platform services are hosted by the SOA4All DSB then they can benefit from its advanced extra features like distribution, automated deployment, lifecycle management, monitoring, etc.
- **Optimal communication performance**. As SCA-based platform services are hosted by the SOA4All DSB, the latter has opportunity to optimize communications between platform services, e.g., for instance when two platform services are hosted on the same SOA4All DSB node then their interactions are done locally without requiring to use WS bindings systematically. Then the performance of communications (i.e.,

[3] For a quick overview of SCA, read Section 5.3.1 of this deliverable and Section 4.1.1 of D1.4.1A [11].

latency, throughput) would certainly be more optimal.

- **Assistance for better reusable software components**. SCA is a component model for building SOA applications. SCA provides means to assist developers to develop highly reusable software components. With SCA, business code is encapsulated into SCA components. Several programming languages can be used to implement SCA components (e.g., Java, BPEL, Spring, etc). Configuration of SCA composite applications is described with a dedicated but simple XML-based language. SCA intents and policies allow developers to express their non-functional concerns, like transactions, security, logging, etc.

5.1.2.2.2 Cons

- **SCA-specific development process**. SCA extends the traditional Web Services process development (e.g., with WSDL and JAX-WS) by introducing some SCA-specific rules to develop composite applications. These few new rules are discussed into Chapter 5.3. Then SOA4All developers will need to learn these new rules. However, the Eclipse IDE already includes a set of SCA-specific plug-ins and wizards to greatly simplify and accelerate the development of SCA composite applications[4]. These SCA-specific tools are included into the official Eclipse Galileo release.

5.1.2.3 The JBI-based Approach

A platform service can be implemented as a JBI-based component. Then this platform service will be hosted on one of the SOA4All DSB nodes provided by EBM WebSourcing, INRIA, or STI.

5.1.2.3.1 Pros

- **Hosting facility provided**. At first, EBM WebSourcing, INRIA and STI will provide SOA4All DSB nodes to host JBI-based platform services. Then SOA4All developers need not to provide and administrate their own public machines. The SOA4All DSB provides tools to facilitate the deployment, management, and monitoring of JBI-based platform services.
- **DSB extra features**. As JBI-based platform services are hosted by the SOA4All DSB then they can benefit from its advanced extra features like distribution, automated deployment, lifecycle management, monitoring, etc. Moreover JBI-based platform services can have a full fine-grain access to and control on JBI features, especially the various communication patterns provided by JBI.
- **Optimal communication performance**. As JBI-based platform services are hosted by the SOA4All DSB, the latter has opportunity to optimize communications between platform services, e.g., for instance when two platform services are hosted on the same SOA4All DSB node then their interactions are done locally without requiring to use WS bindings systematically. Then the performance of communications (i.e., latency, throughput) would certainly be more optimal.

5.1.2.3.2 Cons

- **JBI-specific development process**. JBI extends the traditional Web Services process development (e.g., WSDL, JAX-WS) by introducing some JBI-specific rules to develop JBI components. Then SOA4All developers will need to learn these new rules. However, OW2 PEtALS, the open source distributed Enterprise Service Bus

[4] Have a look to the Eclipse STP/SCA Tools Project at http://www.eclipse.org/stp/sca and http://wiki.eclipse.org/STP/SCA_Component.

(ESB) integrated into the SOA4All DSB, provides the PEtALS Component Development Kit (CDK)[5] to simplify the implementation of JBI components, and a set of Eclipse plug-ins[6] to create and package JBI artefacts for the main PEtALS components.

5.1.2.4 Summary

Table 1 summarizes the pros and cons of the three SOA4All DSB integration approaches.

Table 1: Summary of the three integration approaches

Criteria / Approach	Standalone	SCA-based	JBI-based
Hosting facility	☹	☺	☺
DSB extra features	☹	☺	☺
Communication performance	☹	☺	☺
Assistance for better reusable software components	☹	☺	☺/☹
Development process	☺	☹/☺	☹/☺

☺ = pros; ☹ = cons; ☺/☹ = pros but cons; ☹/☺ = cons but pros

5.1.3 SOA4All Distributed Service Bus Runtime Access

As introduced before, the SOA4All DSB nodes are provided and hosted by eBM WebSourcing, INRIA, and STI.

In order to be aligned with the general SOA guidelines and with the SOA4All architecture delivrables, the DSB nodes need to expose their services (monitoring, management, …) as Web services.

In the current document, the services URL will adopt the following syntax:

http://<HOST>[:<PORT>]/<PATH>/<SERVICE>

Table 2. List of SOA4All DSB Nodes

Provider	Host	Port
eBM WebSourcing	**SOA4All.ebmwebsourcing.com**	**-**
INRIA	**-**	**-**
STI	**-**	**-**

Note: The service consumers' developers need to externalize all the URLs used in

[5] Available at http://petals.ow2.org/download-petals-esb.html

[6] Available at http://petals.ow2.org/download-tools.html

their modules in order to avoid compilation when DSB node API URL will be updated.

5.2 Standalone Platform Services

This chapter provides detailed guidelines on the standalone approach to plug SOA4All Platform Services to the SOA4All DSB.

As a reminder, standalone services means that these Web services will be hosted and managed on SOA4All partner's nodes. The current chapter introduces how these services can be plugged into the SOA4All DSB using the SOA4All management API and is not a guideline on how to design and implement Web services.

5.2.1 Standalone Management API

OW2-PEtALS ESB provides connectors to bind external services to the bus and to expose bus services as external services. This is highly JBI dependent and does not fill the SOA4All management API guidelines.

The OW2-PEtALS ESB is extended to provide a SOAP-based management API, which will be used to build the SOA4All management API. The current operations are defined in the next sections.

5.2.1.1 Service Binder Service

This Web service is used to bind standalone services to the bus and is defined at *http://<HOST>[:<PORT>]/dsb/management/ServiceBinder?wsdl*.

5.2.1.1.1 bindWebService

Binds the Web service defined in the WSDL description to the local PEtALS ESB node. As a result, each message sent to the internal PEtALS endpoint will be forwarded to the real standalone Web service.

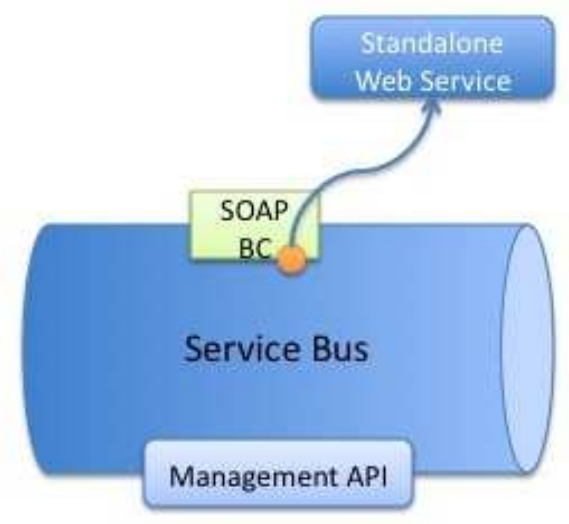


Figure 13: Binding an External Standalone Web Service to the DSB

5.2.1.1.2 unbindWebService

Unbinds a previously bound Web service. The PEtALS endpoint is undeployed from the bus.

5.2.1.1.3 getBoundWebServices

Returns the list of all Web services bound to the service bus.

5.2.1.1.4 proxifyWebService

Same than the `bindWebService` operation and additionally exposes the PEtALS endpoint as a PEtALS Web service. Each SOAP message sent to the PEtALS Web service will be routed to the PEtALS provider endpoint and finally sent to the standalone Web service.

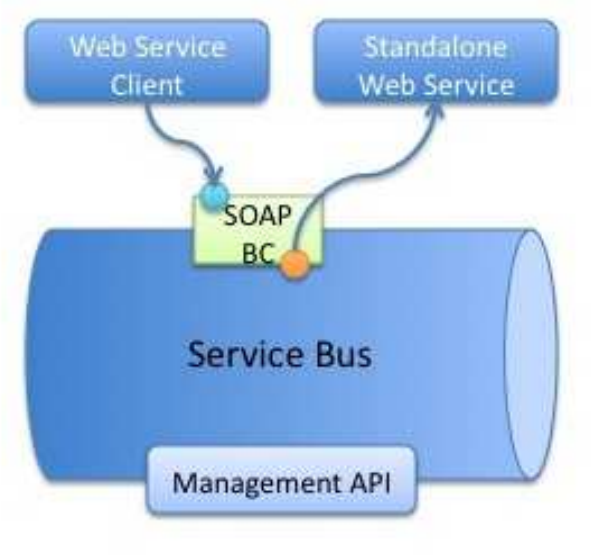


Figure 14: Proxifying an External Standalone Web Service with the DSB

A standalone Web service hosted at

- *http://SOA4All.partner.org/service/PartnerService*

will be proxified by the Service Bus and exposed at

- *http://<HOST>[:<PORT>]/petals/services/PartnerServiceProxy*

finally, its WSDL description will also be available at

- *http://<HOST>[:<PORT>]/petals/services/PartnerService**Proxy**?wsdl*

Note: Due to a limitation in the PEtALS SOAP Binding Component, the proxified WSDL port address may not be the good one. Generated clients must be updated to fix this bug.

5.2.1.1.5 unproxifyWebService

Unproxify a previously proxified Web service.

5.2.1.1.6 getProxifiedWebServices

Returns the list of all Web services proxified by the DSB.

5.3 SCA-based Platform Services

This chapter provides detailed guidelines on the SCA-based approach to plug SOA4All Platform Services to the SOA4All DSB.

5.3.1 Overview of SCA

5.3.1.1 What SCA is

Service Component Architecture (SCA) is a set of specifications initially defined by the Open

Service Oriented Architecture (OSOA) collaboration[7], a set of major companies of the software industry including IBM, IONA, Oracle, SAP, Sun and TIBCO, and now under standardization by the OASIS consortium[8]. SCA defines a component-based model for building service-oriented applications and systems (SOA). As shown in **Error! Reference source not found.**, the SCA model is agnostic against (i.e., independent of) technologies used for defining interfaces (WSDL, Java interfaces, etc.), to implement components (C/C++, BPEL, Java, Spring, EJB, OSGi, COBOL, etc.), to make them to communicate across the network (Web Service, JMS, Java RMI, etc.), and to apply non-functional properties (transactional, security, etc.). Then SCA facilitates the integration of SOA-based applications and systems.

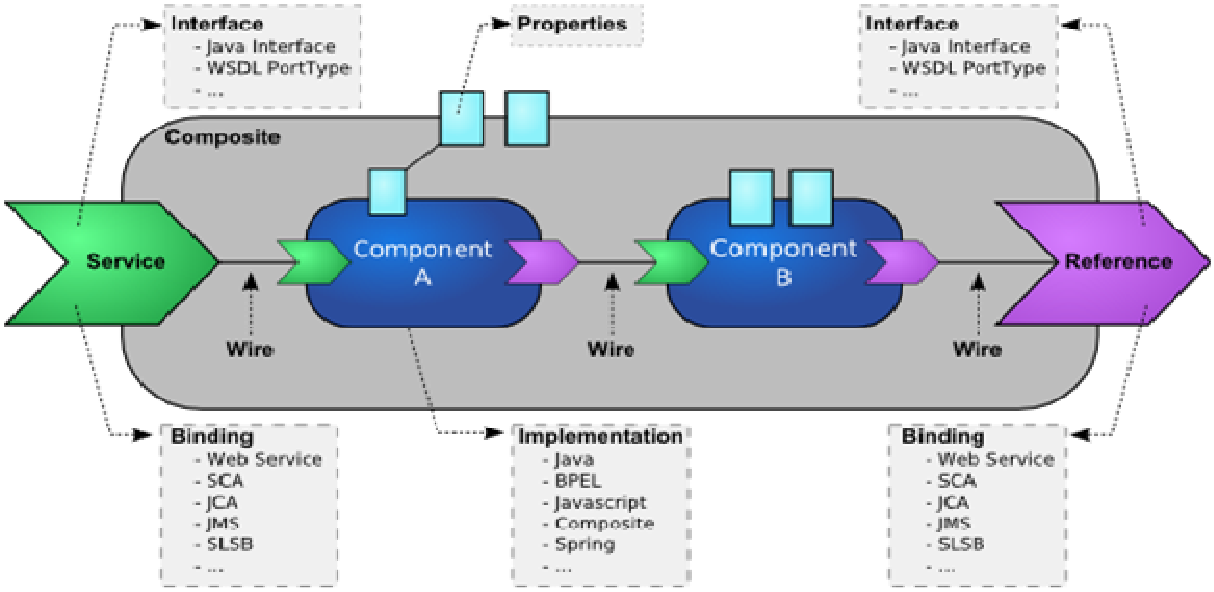


Figure 15: The Service Component Architecture (SCA) Model

The main SCA specification defines the SCA Assembly Model [14], an XML-based language used to describe SCA-based applications or composites. Its key concepts are:

- **Binding** – An SCA binding identifies the protocol used to export an SCA service or import an SCA reference to/from the rest of the world. SCA is extensible to support various forms of bindings. By default, SCA supports Web Services as bindings.
- **Component** – An SCA component encapsulates a running piece of software. Each component has a name, exposes a set of SCA services, has a set of SCA properties, is implemented by one SCA implementation, and requires a set of SCA references.
- **Composite** – An SCA composite makes a set of strongly coupled components to work together. Each composite has a name, exposes a set of SCA services, has a set of SCA properties, is composed of a set of SCA components and wires, and requires a set of SCA references.
- **Implementation** – An SCA implementation defines the implementation of an SCA

[7] See at http://www.osoa.org

[8] See at http://www.oasis-opencsa.org

component. SCA is extensible to support various forms of component implementations. An SCA component can be implemented by an SCA composite, a Java class, a BPEL process, a set of C functions, etc.

- **Interface** – An SCA interface defines the signature of operations/methods provided by an SCA service or reference. SCA is extensible to support various languages to express signatures. By default, SCA supports WSDL and Java as interface definition languages.
- **Property** – An SCA property identifies a configurable business property of a component or composite. Each property has a name, a type, and a value.
- **Reference** – An SCA reference identifies a dependency of a component/composite to a service. Each reference has a name, is defined by an SCA interface, and can be bound via SCA bindings.
- **Service** – An SCA service is an access point to invoke an SCA component or composite. Each service has a name, is defined by an SCA interface, and can be exported through SCA bindings.
- **Wire** – An SCA wire identifies a communication path from an SCA reference source to an SCA service target.

As depicted in , the SOA4All DSB supports SCA thanks to an SCA Service Engine running on top of the OW2 PEtALS ESB. This service engine supports the SCA Assembly Model and is built on top of the OW2 FraSCAti SCA runtime[9].

5.3.1.2 What Eclipse STP/SCA is

The Eclipse STP/SCA Tools project[10] provides a set of tools for simplifying the design and speeding up the development of SCA-based applications. This includes a graphical composer, various editors (XML-based, tree-based, form-based) and several plug-ins for specifying, developing, assembling, and deploying SCA composites. Eclipse STP/SCA Tools are included into the official Eclipse Galileo release.

[9] See at http://frascati.ow2.org

[10] See at http://www.eclipse.org/stp/sca and http://wiki.eclipse.org/STP/SCA_Component.

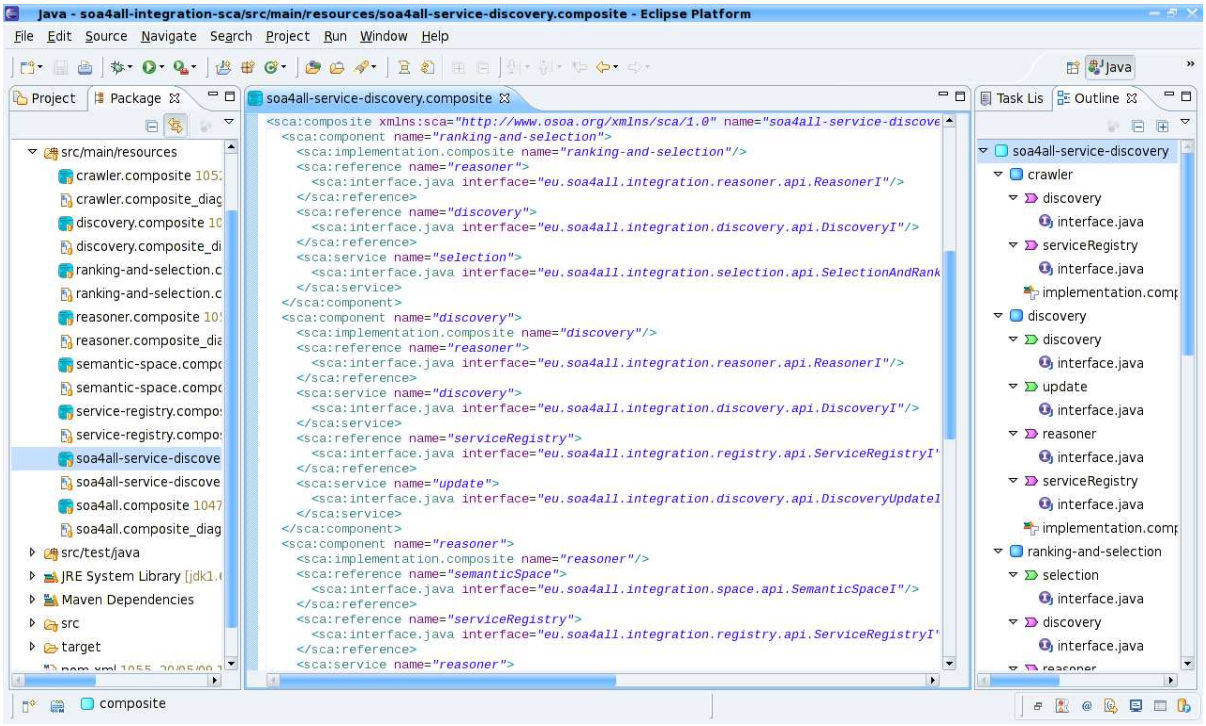


Figure 16: The Eclipse STP/SCA XML Editor

Error! Reference source not found. illustrates the Eclipse STP/SCA XML-based editor (the middle panel) and the Eclipse STP/SCA tree-based editor (the right panel), both are driven by the XML Schemas defined by the SCA Assembly Model. **Error! Reference source not found.** illustrates the Eclipse STP/SCA graphical composer allowing one to graphically "draw" SCA composites. The example used in both figures is an SCA composite representing the SOA4All Service Discovery Functional Process detailed into next sections.

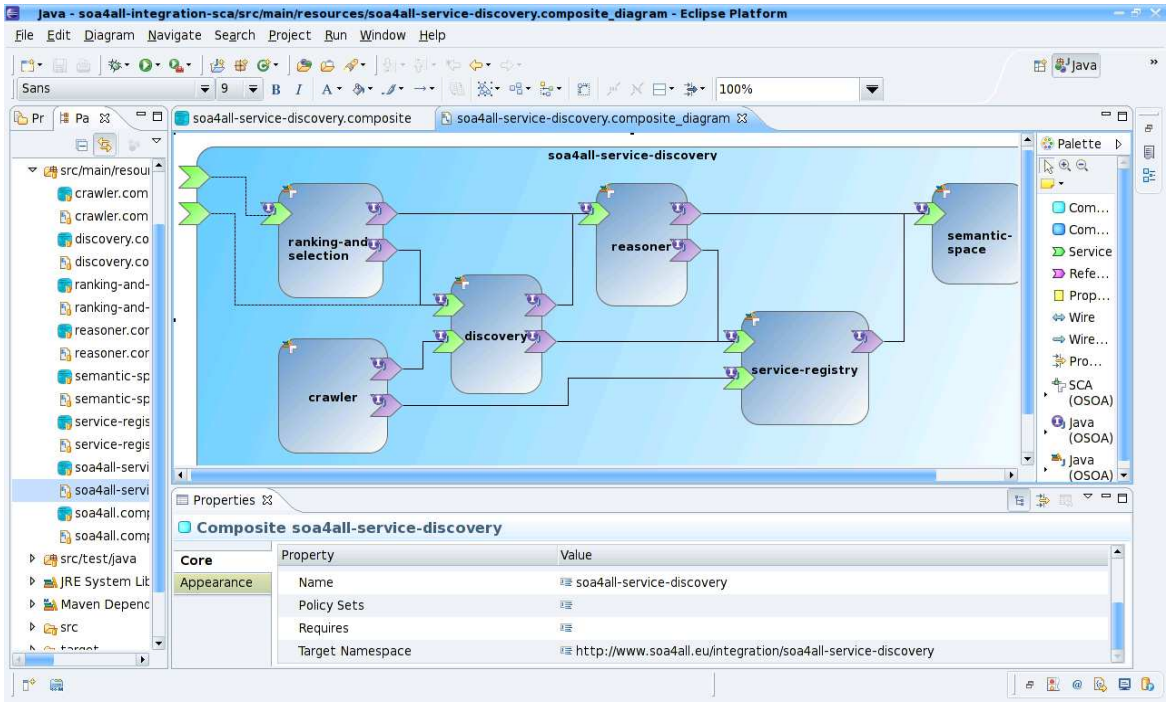


Figure 17: The Eclipse STP/SCA Composer

5.3.2 Building SCA-based Platform Services

In order to illustrate how to develop SOA4All Platform Services as SCA-based composites, this section is based on the SOA4All Service Discovery Functional Process. This process is described into Section 7.4.3 of D1.4.1A [11] and depicted into **Error! Reference source not found.** taken from the SOA4All Architecture presented during the 2nd project review [2].

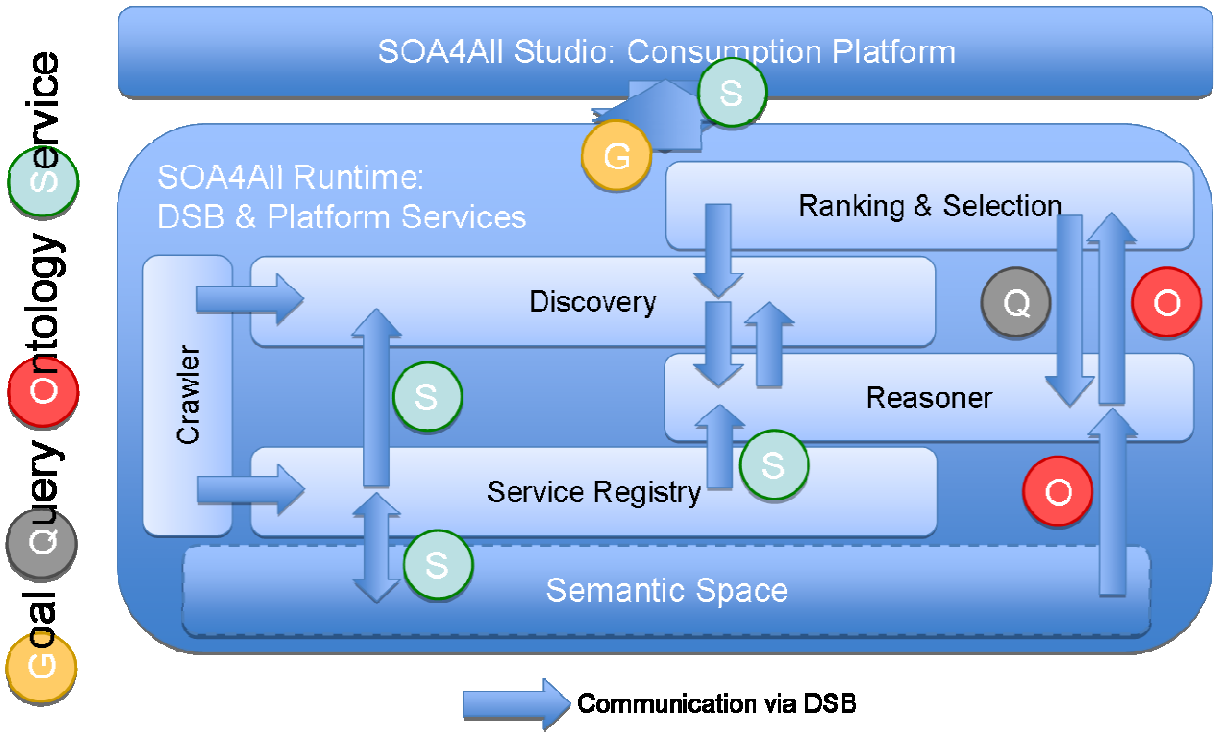


Figure 18: The SOA4All Service Discovery Functional Process

The SOA4All Service Discovery Functional Process involves five SOA4All Platform Services (Ranking&Selection, Discovery, Crawler, Reasoner, Service Registry) and the SOA4All Semantic Space. Each rectangle represents a SOA4All component (semantic space, platform services, or studio). The arrows represent communications between SOA4All components and transported by the SOA4All DSB transparently. The circles represent the communication objects as defined into Section 6 of D1.4.1A [11].

A prototype of the SCA-based implementation of the SOA4All Service Discovery Functional Process is available into the SOA4All SVN at the following URL:

https://svn.sti2.at/SOA4All/trunk/SOA4All-integration/SOA4All-integration-sca

This implementation contains all the SCA composites shown into Section 5.3.2.1, Java interfaces and implementation classes of each involved SOA4All components. Of course, these interfaces and classes are just skeletons for illustration purposes, and must be completed by SOA4All components' developers.

5.3.2.1 Designing SCA-based Platform Services

At a high level of abstraction, we can consider that the whole SOA4All platform is an SCA composite. For instance, **Error! Reference source not found.** shows an SCA composite composed of two components: one component encapsulates the SOA4All Studio Consumption Platform and has two SCA references wired to two SCA services provided by another component encapsulating the SOA4All Service Discovery Functional Process.

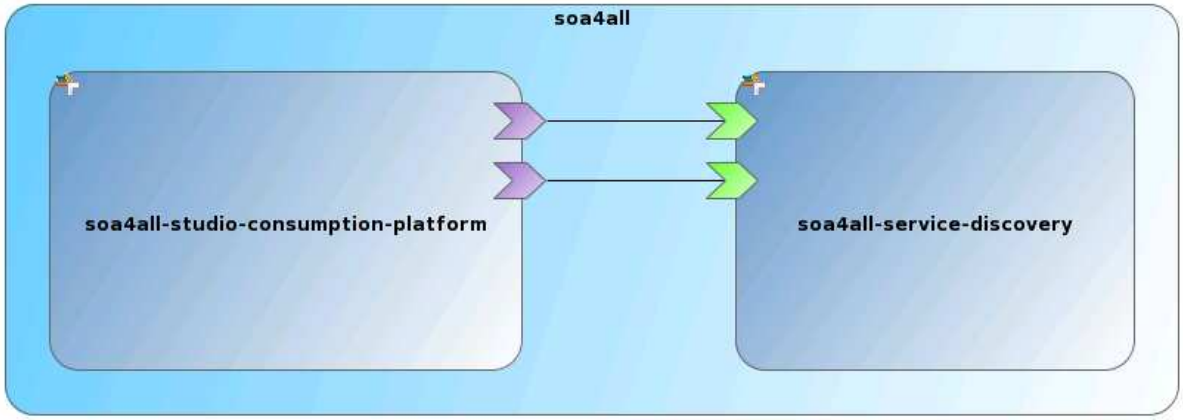


Figure 19: The SOA4All SCA composite

The SOA4All Service Discovery component can be itself implemented by an SCA composite. This composite shown in **Error! Reference source not found.** is composed of six components representing the Ranking&Selection, Crawler, Discovery, Reasoner, ServiceRegistry SOA4All platform services, and the semantic space. Each arrow of **Error! Reference source not found.** is represented by an SCA wire, i.e., a communication path from an SCA reference to an SCA service. Each SCA reference represents a dependency from its associated SOA4All component to an SCA service provided by another SOA4All component. A SOA4All component can have several SCA references. For instance, the SOA4All `reasoner` component requires both `service-registry` and `semantic-space` components in order to realize its functionalities. Each SCA service represents a set of functionalities offered by a SOA4All component. A SOA4All component can provide several SCA services. For instance, the `discovery` component provides a service used by the `ranking-and-selection` component and another service used by the `crawler` component. Defining several services for a component is a design choice allowing us to better identify the various roles that can play a component in the whole architecture, and avoiding to define for each component one big interface containing all its operations. Here we apply the principle of separation of concerns at the application design level.

Each of the six components of the `SOA4All-service-discovery` SCA composite can itself implemented as an SCA composite. This hierarchical approach allows us to go from a high-level global view of the whole architecture (see **Error! Reference source not found.**) to a fine-grain view of its implementation (see **Error! Reference source not found.** to **Error! Reference source not found.**). Here we decide to stop the recursivity: Each of next six SOA4All composites just contains one SCA component implemented by a Java class. Of course, each provider can decide to recursively decompose its SOA4All platform service as a set of more than one SCA components.

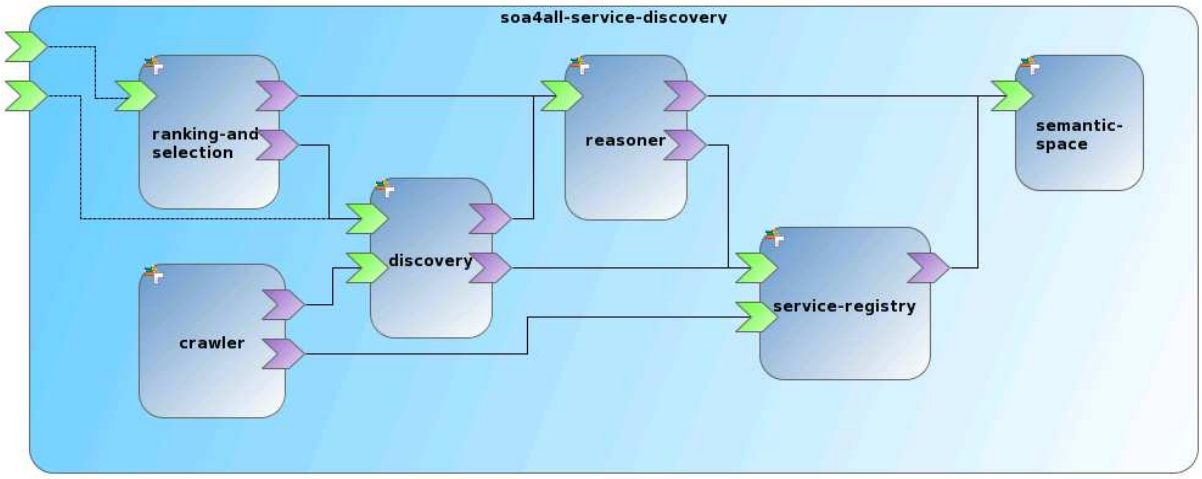


Figure 20: The SOA4All-service-discovery SCA Composite

Error! Reference source not found. represents the semantic-space SCA composite containing one SCA component, which provides an SCA service defined by a Java interface, provides a configurable SCA property, and is implemented by a Java class. The XML-based document describing this SCA composite is provided later in Section 5.3.2.4 **Error! Reference source not found.**. The motivations to apply SCA on the SOA4All Semantic Space component are given in Section 4.1.4 of D1.4.1A [11]. The Java interface of this component is the one defined into D1.3.2A [12].

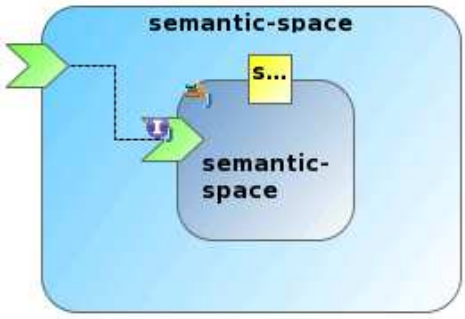


Figure 21: The semantic-space SCA Composite

Error! Reference source not found. to **Error! Reference source not found.** represent `service-registry`, `reasoner`, `discovery`, `ranking-and-selection`, and `crawler` SCA composites. These composites must be enhanced by their respective provider, i.e., decomposing the platform service into several SCA components, adding new SCA services, and adding new configurable references and properties.

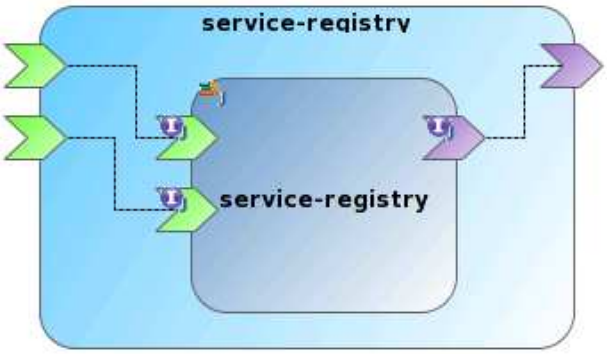


Figure 22: The service-registry SCA Composite

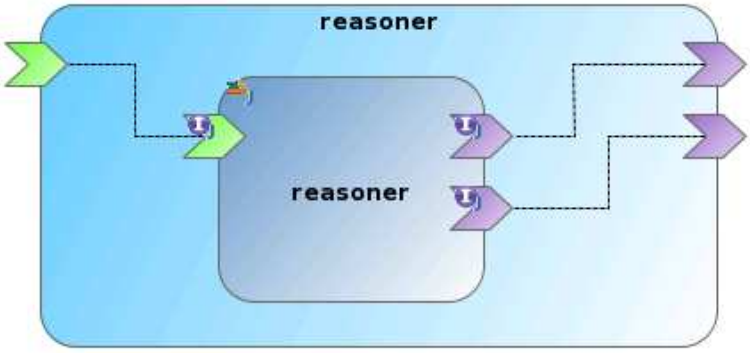


Figure 23: The reasoner SCA Composite

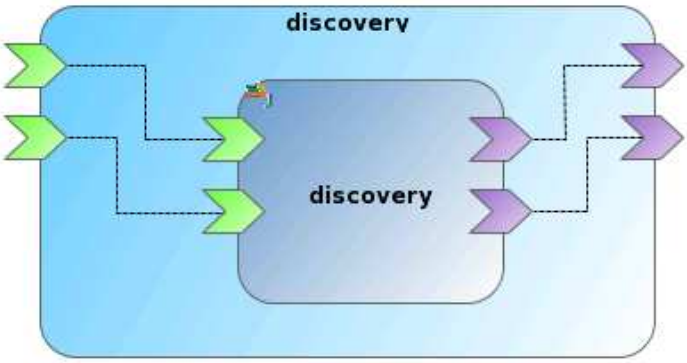


Figure 24: The discovery SCA Composite

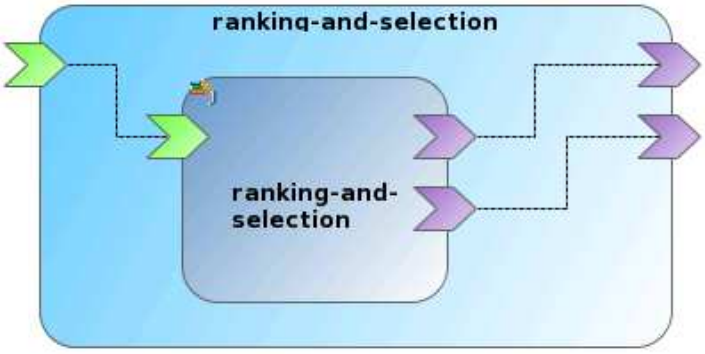


Figure 25: The ranking-and-selection SCA Composite

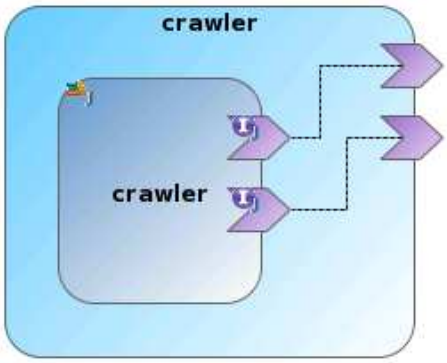


Figure 26: The crawler SCA Composite

5.3.2.2 Defining SCA-based Platform Service Interfaces

By nature, SCA can support various languages to define service interfaces. By default, SCA supports WSDL port types and Java interfaces. We recommend using Java instead of WSDL port types to SOA4All developers, as defining Java interfaces is more concise and avoiding errors. Moreover, the JAX-WS specification [13] provides a set of Java annotations allowing us to annotate Java interfaces in order to produce their equivalent WSDL port types automatically.

```
1     package eu.SOA4All.integration.space.api;
2     import java.util.Set;
3     import javax.jws.WebMethod;
4     import javax.jws.WebService;
5     import org.openrdf.model.Statement;
6     import eu.SOA4All.dsb.space.query.Query;
7     @WebService
8     public interface SemanticSpaceI {
9       @WebMethod public Set<Statement> query(Query q);
10      ...
11    }
```

Figure 27: The SemanticSpaceI Java Interface

Error! Reference source not found., **Error! Reference source not found.**, and **Error! Reference source not found.** illustrate how to use JAX-WS to annotate SOA4All Java interfaces. `@WebService` is used to annotate Java interfaces in order to expose them as WSDL port types and `@WebMethod` is used to annotate Java methods in order to expose them as WSDL operations. These two Java annotations have a set of associated attributes allowing us to configure the mapping to WSDL in a fine-grain way (read [13] for more details).

```
1     package eu.SOA4All.integration.discovery.api;
2     import java.util.Set;
3     import javax.jws.WebMethod;
4     import javax.jws.WebService;
5     import eu.SOA4All.integration.fake.Goal;
6     import eu.SOA4All.integration.fake.ServiceDescription;
7     @WebService
8     public interface DiscoveryI {
9       @WebMethod
10      public Set<ServiceDescription> discover(Goal goal);
11    }
```

Figure 28: The DiscoveryI Java Interface

```
1     package eu.SOA4All.integration.registry.api;
2     import java.util.Set;
3     import javax.jws.WebMethod;
4     import javax.jws.WebService;
5     import eu.SOA4All.dsb.space.query.Query;
6     import eu.SOA4All.integration.fake.ServiceDescription;
7     @WebService
8     public interface ServiceRegistryI {
9       @WebMethod
10      public Set<ServiceDescription> getServiceDescription(Query query);
11    }
```

Figure 29: The ServiceRegistryI Java Interface

Let us note that these three previous interfaces are not normative[11], and must be fully defined by their associated providers (WP1 for `SemanticSpaceI`, WP5 for `DiscoveryI` and `ServiceRegistryI`). **SOA4All developers should just define Java interfaces annotated with JAX-WS**.

5.3.2.3 Implementing SCA-based Platform Services

An SCA component can be implemented as an SCA composite (see **Error! Reference source not found.**) or via a Java class (see **Error! Reference source not found.** to **Error! Reference source not found.**). This section describes the few rules that developers must follow in order to develop Java classes implementing SCA components.

To illustrate these rules, this section provides a skeleton of the Java implementation of three

[11] The goal of this deliverable is not to specify the name of each Java interface and the signature of their Java methods. This is the purpose of WP3 to WP6 deliverables.

SOA4All components: `semantic-space` (see **Error! Reference source not found.**), `service-registry` (see **Error! Reference source not found.**) and `discovery` (see **Error! Reference source not found.**).

A Java class implementing an SCA component must implement:

1. **a public default constructor**. This public constructor without parameters is called by the SCA framework to instantiate the Java class. Let us note that a Java class without any declared constructor has an implicit public default constructor, which is the case for Java classes `SemanticSpaceImpl` (see **Error! Reference source not found.**), `ServiceRegistryImpl` (see **Error! Reference source not found.**), and `DiscoveryImpl` (see **Error! Reference source not found.**).
2. **all the interfaces of all the SCA services of its enclosing SCA component**. For instance, the `SemanticSpaceImpl` class implements the `SemanticSpaceI` interface (lines 3 and 6-8) as its enclosing SCA component provides one SCA service (see **Error! Reference source not found.**). But each class `ServiceRegistryImpl` and `DiscoveryImpl` implements two Java interfaces (lines 3 and 6-9 in **Error! Reference source not found.**, lines 3 and 7-11 in **Error! Reference source not found.**), as their respective SCA component provides two SCA services (see **Error! Reference source not found.** and **Error! Reference source not found.**).
3. **a public setter method for each SCA reference and property of its enclosing SCA component**. For instance, the `SemanticSpaceImpl` class contains the `setStorage` method (line 5 in **Error! Reference source not found.**) to be notified by the SCA framework of the value of the SCA property named `storage` and of type `string`. The concrete value of this property will be set in an SCA composite descriptor (see Section **5.3.2.4**). In the `ServiceRegistryImpl` class, the method `setSemanticSpace` is used by the SCA framework to set the value of the SCA reference named `semanticSpace` and of interface `SemanticSpaceI`. The `DiscoveryImpl` class has two public setter methods for setting the SCA references named `reasoner` and `serviceRegistry`. This SCA programming pattern is in fact the same as used by JavaBeans and Spring component models.

These three simple rules are enough to implement most of components. Their main advantage is then that SCA is not intrusive into Java classes, i.e., a SOA4All developer can implement a Java class without requiring to know SCA in depth but just the three simple previous rules.

However, advanced component implementations can require to be SCA-aware (e.g., for being notified of the life cycle of components). For this purpose, some SCA-specific Java annotations and API are specified in [15] and [14][12].

[12] These SCA-specific Java annotations and API would be discussed in a next version of this deliverable according to SOA4All development requirements.

```
1     package eu.SOA4All.integration.space.lib;
2     import . . .
3     public class SemanticSpaceImpl implements SemanticSpaceI {
4       // Setter methods for configuration
5       public void setStorage(String s) { . . . }
6       // Implementation of the SemanticSpaceI interface
7       public Set<Statement> query(Query q) { . . . }
8       . . .
9     }
```

Figure 30: The Java Implementation of the Semantic Space Component

```
1     package eu.SOA4All.integration.registry.lib;
2     import . . .
3     public class ServiceRegistryImpl implements ServiceRegistryI,
      ServiceRegistryUpdateI {
4       // Setter methods for configuration
5       public void setSemanticSpace(SemanticSpaceI s) { . . . }
6       // Implementation of the ServiceRegistryI interface
7       public Set<ServiceDescription>
        getServiceDescription(Query query) { . . . }
8       // Implementation of the ServiceRegistryUpdateI interface
9       . . .
10    }
```

Figure 31: The Java Implementation of the Service Registry Component

```
1     package eu.SOA4All.integration.discovery.lib;
2     import . . .
3     public class DiscoveryImpl implements DiscoveryI,
      DiscoveryUpdateI {
4       // Setter methods for configuration
5       public void setReasoner(ReasonerI r) { . . . }
6       public void setServiceRegistry(ServiceRegistryI sr) {...}
7       // Implementation of the DiscoveryI interface
8       public Set<ServiceDescription> discover(Goal goal) { ... }
9       // Implementation of the DiscoveryUpdateI interface
10      . . .
11    }
```

Figure 32: The Java Implementation of the Discovery Component

5.3.2.4 Configuring SCA-based Platform Services

This section presents concrete examples of the SCA XML-based language allowing us to define SCA composites, configure SCA properties, wire SCA references to services, and export SCA services as Web Service endpoints. SOA4All architects can write these XML documents with a simple text editor directly, or use advanced graphical editors provided by Eclipse STP/SCA Tools (see Section 5.3.1.2).

```
1     <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
                 name="semantic-space"
                 targetNamespace="http://www.SOA4All.eu">
2       <service name="semantic-space"
                 promote="semantic-space/semanticSpace"/>
3       <component name="semantic-space">
4         <implementation.java
            class="eu.SOA4All.integration.space.lib.SemanticSpaceImpl"/>
5         <property name="storage">/semantic-space-repository</property>
6         <service name="semanticSpace">
7           <interface.java
             interface="eu.SOA4All.integration.space.api.SemanticSpaceI"/>
8         </service>
9       </component>
10    </composite>
```

Figure 33: The XML-based semantic-space SCA Composite

Error! Reference source not found. defines the SCA composite for the SOA4All Semantic Space component as shown in **Error! Reference source not found.**. The XML Schema of SCA is available at `http://www.osoa.org/xmlns/sca/1.0` (line 1). The name and target namespace of this composite are `semantic-space` and `http://www.SOA4All.eu` respectively (line 1). This composite exports an SCA service named `semantic-space` and promoting the SCA service named `semanticSpace` of the SCA component named `semantic-space` (line 2). This composite contains an SCA component named `semantic-space` (line 3). This component is implemented by a Java class (line 4). It has the SCA property named `storage`, which is configured with the string value "`/semantic-space-repository`" (line 5). It provides an SCA service named `semanticSpace` (line 6) and defined by a Java interface (line 7). Let us note that the Java interface and class were defined previously in **Error! Reference source not found.** and **Error! Reference source not found.** respectively.

```
1     <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
                 name="service-registry"
                 targetNamespace="http://www.SOA4All.eu">
2       <service name="registry" promote="service-registry/registry"/>
3       <service name="update" promote="service-registry/update"/>
4       <reference name="semantic-space"
                   promote="service-registry/semantic-space"/>
5       <component name="service-registry">
6         <implementation.java
           class="eu.SOA4All.integration.registry.lib.ServiceRegistryImpl"/>
7         <service name="registry">
8           <interface.java interface=
                    "eu.SOA4All.integration.registry.api.ServiceRegistryI"/>
9         </service>
10        <service name="update">
11          <interface.java interface=
              "eu.SOA4All.integration.registry.api.ServiceRegistryUpdateI"/>
12        </service>
13        <reference name="semanticSpace">
14          <interface.java interface=
                       "eu.SOA4All.integration.space.api.SemanticSpaceI"/>
15        </reference>
16      </component>
17    </composite>
```

Figure 34: The XML-based service-registry SCA Composite

Error! Reference source not found. defines the SCA composite for the SOA4All Service Registry Platform Service as shown in **Error! Reference source not found.** (line 1). This composite contains one SCA component (line 5). This component is implemented by one Java class (line 6), which was defined in **Error! Reference source not found.**. It provides two SCA services each defined by a Java interface (lines 7-12). It requires an SCA reference to the semantic space (lines 13-15). Both services and the reference of this component are promoted at the level of its enclosing composite (lines 2-4).

```
1     <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
                 name="discovery"
                 targetNamespace="http://www.SOA4All.eu">
2       <service name="discovery" promote="discovery/discovery"/>
3       <service name="update" promote="discovery/update"/>
4       <reference name="reasoner" promote="discovery/reasoner"/>
5       <reference name="service-registry"
                   promote="discovery/serviceRegistry"/>
6       <component name="discovery">
7         <implementation.java
               class="eu.SOA4All.integration.discovery.lib.DiscoveryImpl"/>
8         <service name="discovery">
9           <interface.java interface=
10                      "eu.SOA4All.integration.discovery.api.DiscoveryI"/>
11        </service>
12        <service name="update">
13          <interface.java interface=
                    "eu.SOA4All.integration.discovery.api.DiscoveryUpdateI"/>
14        </service>
15        <reference name="reasoner">
16          <interface.java interface=
                        "eu.SOA4All.integration.reasoner.api.ReasonerI"/>
17        </reference>
18        <reference name="serviceRegistry">
19          <interface.java interface=
20                  "eu.SOA4All.integration.registry.api.ServiceRegistryI"/>
21        </reference>
22      </component>
23    </composite>
```

Figure 35: The XML-based discovery SCA Composite

Error! Reference source not found. defines the SCA composite for the SOA4All Discovery Platform Service as shown in **Error! Reference source not found.** (line 1). This composite contains one SCA component (line 6). This component is implemented by one Java class (line 7), which was defined in **Error! Reference source not found.**. It provides two SCA services and requires two SCA references, each defined by a Java interface (lines 8-21). All the services and references of this component are promoted at the level of its enclosing composite (lines 2-5).

Error! Reference source not found. defines the SCA composite for the SOA4All Service Discovery Functional Process as shown in **Error! Reference source not found.** (line 1). This composite contains six SCA components implemented themselves as SCA composites (lines 8-25). Let us note that three of these composites (`semantic-space`, `service-registry`, and `discovery`) were presented in previous **Error! Reference source not found.**, **Error! Reference source not found.**, and **Error! Reference source not found.** respectively. All the references of the enclosed components are wired to appropriate services (lines 26-34). This composite exports two SCA services promoted from the `ranking-and-selection` and `discovery` SCA components (lines 2-7). Finally, these two composite services are exposed as Web Services and will be accessible by clients at the following Web addresses `http://localhost:8080/Selection` and `http://localhost:8080/Discovery` (lines 3 and 6).

```
1     <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
                 name="SOA4All-service-discovery"
                 targetNamespace="http://www.SOA4All.eu">
2       <service name="selection"
                 promote="ranking-and-selection/selection">
3         <binding.ws uri="http://localhost:8080/Selection"/>
4       </service>
5       <service name="discovery" promote="discovery/discovery">
6           <binding.ws uri="http://localhost:8080/Discovery"/>
7       <service>
8       <component name="ranking-and-selection">
9         <implementation.composite name="ranking-and-selection"/>
10      </component>
11      <component name="discovery">
12        <implementation.composite name="discovery"/>
13      </component>
14      <component name="reasoner">
15        <implementation.composite name="reasoner"/>
16      </component>
17      <component name="service-registry">
18        <implementation.composite name="service-registry"/>
19      </component>
20      <component name="crawler">
21        <implementation.composite name="crawler"/>
22      </component>
23      <component name="semantic-space">
24        <implementation.composite name="semantic-space"/>
25      </component>
26      <wire source="service-registry/semantic-space"
              target="semantic-space/semantic-space"/>
27      <wire source="discovery/service-registry"
              target="service-registry/registry"/>
28      <wire source="reasoner/semantic-space"
              target="semantic-space/semantic-space"/>
29      <wire source="reasoner/service-registry"
              target="service-registry/registry"/>
30      <wire source="discovery/reasoner"
              target="reasoner/reasoner"/>
31      <wire source="ranking-and-selection/reasoner"
              target="reasoner/reasoner"/>
32      <wire source="ranking-and-selection/discovery"
              target="discovery/discovery"/>
33      <wire source="crawler/discovery"
              target="discovery/update"/>
34      <wire source="crawler/service-registry"
              target="service-registry/update"/>
35    </composite>
```

Figure 36: The XML-based SOA4All-service-discovery SCA Composite

5.3.2.5 Binding SCA-based Platform Services to External Web Services

The previous section presented how SCA-based platform services can be exposed as Web Services by configuring XML-based SCA composites only. This section illustrates how an SCA-based platform service can be bound to and can invoke an external (certainly non-SCA-based) Web Service.

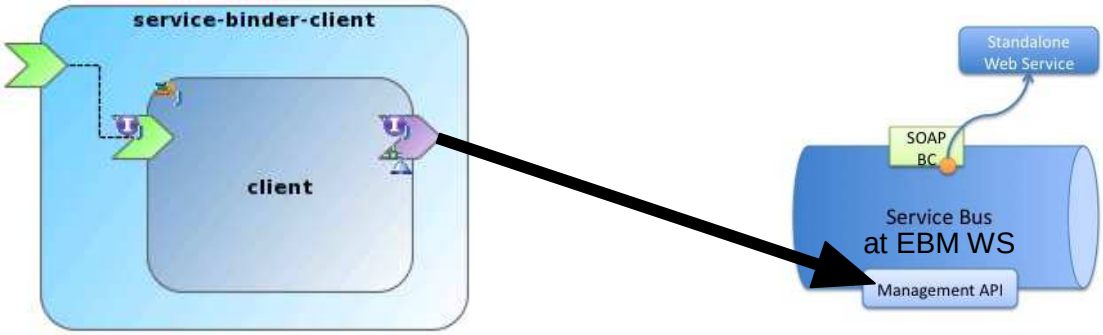


Figure 37: An SCA-based Client of the DBS Service Binder Service

Error! Reference source not found. illustrates an SCA-based application (at the left part) bound to the Service Binder Web Service deployed on the DSB node at EBM WebSourcing (at the right part). This SCA composite is composed of an SCA component named `client`, implemented in Java and having an SCA reference bound to the Web Service (the black arrow in **Error! Reference source not found.**). This Web Service is available at the following URL[13]:

http://SOA4All.ebmwebsourcing.com/dsb/management/ServiceBinder?wsdl

```
1     <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
                 name="service-binder-client"
                 xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
                 targetNamespace="http://www.SOA4All.eu">
2       <service name="run" promote="client/run"/>
3       <component name="client">
4         <implementation.java
            class="eu.SOA4All.integration.servicebinder.ServiceBinderClient"/>
5         <service name="run">
6           <interface.java interface="java.lang.Runnable"/>
7         </service>
8         <reference name="serviceBinder">
9           <interface.java
        interface="org.ow2.petals.core.kernel.ext.servicebinder.ServiceBinder"/>
10          <binding.ws
wsdli:wsdlLocation="http://SOA4All.ebmwebsourcing.com/dsb/management/ServiceBinder?wsdl"
wsdlElement="http://SOA4All.ebmwebsourcing.com/dsb/management#wsdl.port(ServiceBinderServi
ceService/ServiceBinderServicePort)"/>
11        </reference>
12      </component>
13    </composite>
```

Figure 38: The XML-based service-binder-client SCA Composite

Error! Reference source not found. provides the detailed XML-based description of the `service-binder-client` SCA composite. This composite (defined in line 1) provides the `run` SCA service promoted from its enclosed component (line 2), and contains the `client` SCA component (line 3). This component is implemented by a Java class (line 4) defined in **Error! Reference source not found.**. It provides the `run` SCA service (line 5) defined by the `java.lang.Runnable` Java interface (line 6). This component has an SCA reference

[13] The operations of this Web Service were discussed in Section 5.2.1.1.

named `serviceBinder` (line 8), defined by a Java interface specific to the PEtALS ESB (line 9), and bound to the Service Binder Web Service (line 10). The two attributes of `<binding.ws>` identify the location where the WSDL of the Web Service to use is accessible (`wsdlLocation`), and which its WS port to use (`wsdlElement`).

```
1     package eu.SOA4All.integration.servicebinder;
2     import org.ow2.petals.core.kernel.ext.servicebinder.ServiceBinder;
3     public class ServiceBinderClient implements Runnable {
4       private ServiceBinder binder;
5       // Setter for the SCA 'binder' reference.
6       public void setServiceBinder(ServiceBinder b) {
7         this.binder = b;
8       }
9       // Implementation of the Runnable interface.
10      public void run() {
11        System.out.println("List of services bound to the DSB:");
12        for(String s: binder.getBoundWebServices())
13          System.out.println(" - " + s);
14        System.out.println("List of services proxified via the DSB:");
15        for(String s: binder.getProxifiedWebServices())
16          System.out.println(" - " + s);
17      }
18    }
```

Figure 39: The Java Implementation of the Service Binder Client

Error! Reference source not found. provides the Java implementation of the `client` SCA component of the `service-binder-client` SCA composite. This `ServiceBinderClient` Java class must implement the `Runnable` Java interface (line 3) as the SCA enclosing component provides one SCA service of this Java interface type. This class must implement a setter method for the SCA reference named `serviceBinder` of its SCA enclosing component (lines 6-8). Here, the reference is stored into the `binder` Java field (declared at line 4 and set at line 7). Finally, this class must implement the `run` method of the `Runnable` interface (lines 10-17). Here, the Service Binder Service is invoked to obtain the list of all bound and proxified Web Services (line 12 and 15 respectively).

Let us note that this Java class is not strongly coupled to the Service Binder deployed at EBM WebSourcing. This code can be reused as it to invoke the Service Binder deployed on any DSB node. The configuration must just be done in the XML-based enclosing SCA composite, by setting the attributes of `<binding.ws>`.

5.3.2.6 Testing SCA-based Platform Services

We propose to SOA4All developers to use the JUnit framework and the OW2 FraSCAti runtime for testing their SCA-based platform services on their local machine.

Each Java test case class must:

1. **Extend the `eu.SOA4All.integration.test.SCATestCase` class**, which hides the management of the underlying OW2 FraSCAti runtime.
2. **Implement the `public String getComposite()` method**, which must return the name of the SCA composite to test.
3. **Implement methods annotated with `@org.junit.Test`**, which must do test cases on the SCA composite.
4. **Call the `<T>T getService(Class<T> cl, String serviceName)` method** inherited from the `SCATestCase` class, which returns the SCA service named `serviceName` of interface `cl` of the SCA composite under test.

Their four programming conventions are enough for testing SCA composites as illustrated in **Error! Reference source not found.**. The `ServiceDiscoveryTestCase` class extends the `SCATestCase` class (line 8), returns `SOA4All-service-discovery` as the SCA composite name to test (lines 9-11), and implements a test case method annotated with `@Test` (lines 13-16). Then, this test case gets the `discovery` service of the SCA composite (line 14) and invokes it (line 15).

```
1     package eu.SOA4All.integration.test;
2     import java.util.Set;
3     import eu.SOA4All.integration.fake.Goal;
4     import eu.SOA4All.integration.fake.ServiceDescription;
5     import eu.SOA4All.integration.discovery.api.DiscoveryI;
6     import org.junit.Test;
7
8     public class ServiceDiscoveryTestCase extends SCATestCase {
9       public String getComposite() {
10        return "SOA4All-service-discovery";
11      }
12      @Test
13      public void test() {
14        DiscoveryI discovery =
            getService(DiscoveryI.class, "discovery");
15        Set<ServiceDescription> result =
            discovery.discover(new Goal());
16      }
17    }
```

Figure 40: A Java-based Test Case Example

SOA4All developers can also use other testing frameworks like the Web Services Testing tool soapUI[14]. This tool supports functional and load testing of any REST and Web Service. **Error! Reference source not found.** illustrates the usage of soapUI to invoke the SOA4All Semantic Space component.

[14] Available at http://www.soapui.org.

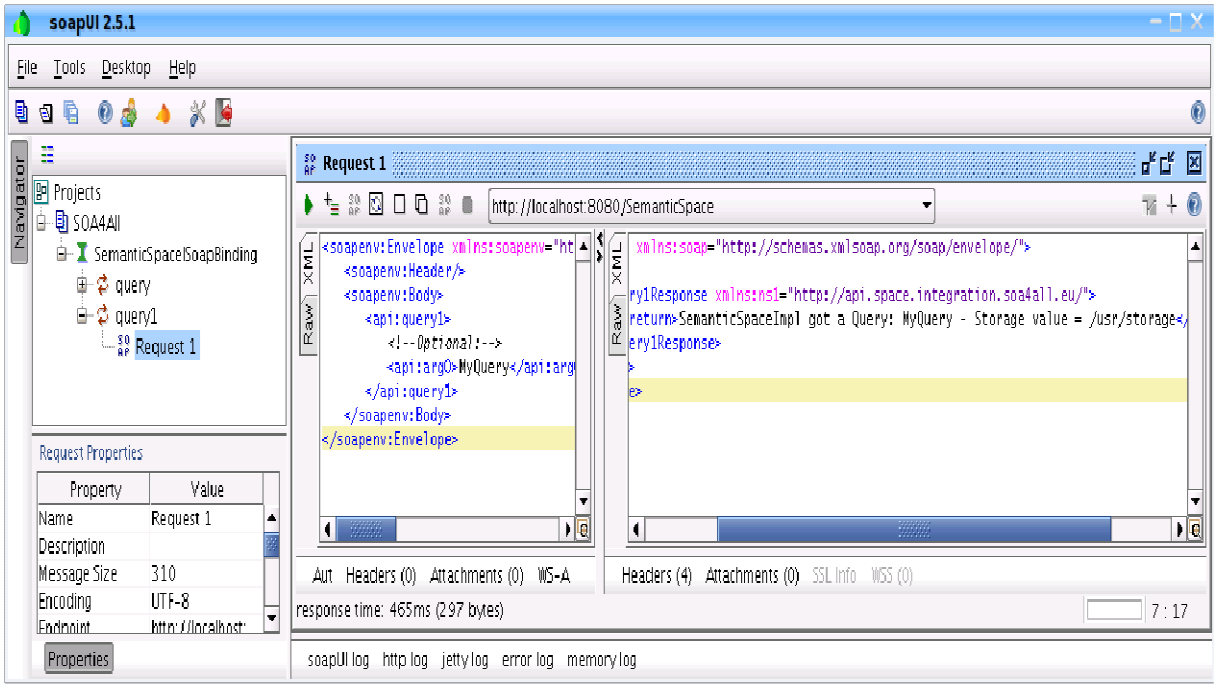


Figure 41: Testing SCA-based Platform Services with soapUI

5.3.2.7 Managing SCA-based Platform Services

As shown in previous sections, SCA is an appropriate technology agnostic standard for designing, developing, implementing, configuring, and deploying SOA4All Platform Services, and more generally any service-oriented application. However, SCA does not define standard means for runtime manageability (including introspection and reconfiguration) of running SOA applications and of their supporting environment. Hopefully the FraSCAti runtime, part of the SOA4All DSB implementation architecture (see), brings runtime management features to SCA [4] via an API for developers and a GUI for end-users (called FraSCAti Explorer).

Error! Reference source not found. illustrates the introspection of a running instance of the `SOA4All-service-discovery` SCA composite. With the left panel, users could discover running SCA composites and their SCA components/services/references/properties. Right panels provide more detailed introspection information. **Error! Reference source not found.** shows the list of SCA references, their interface types, and the SCA service to which they are wired to. **Error! Reference source not found.** illustrates panels to introspect and reconfigure SCA properties and SCA binding.

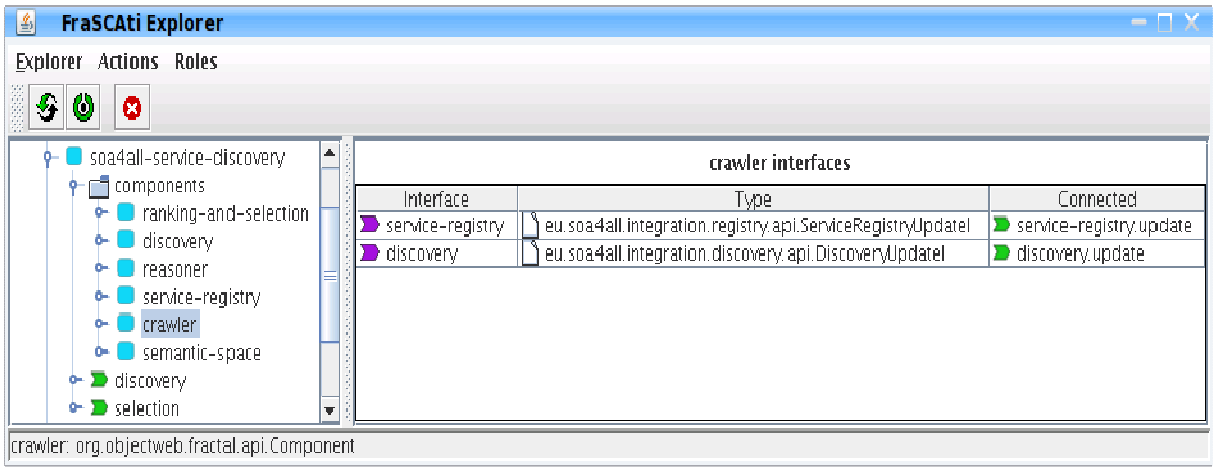


Figure 42: Introspecting SCA-based Platform Services

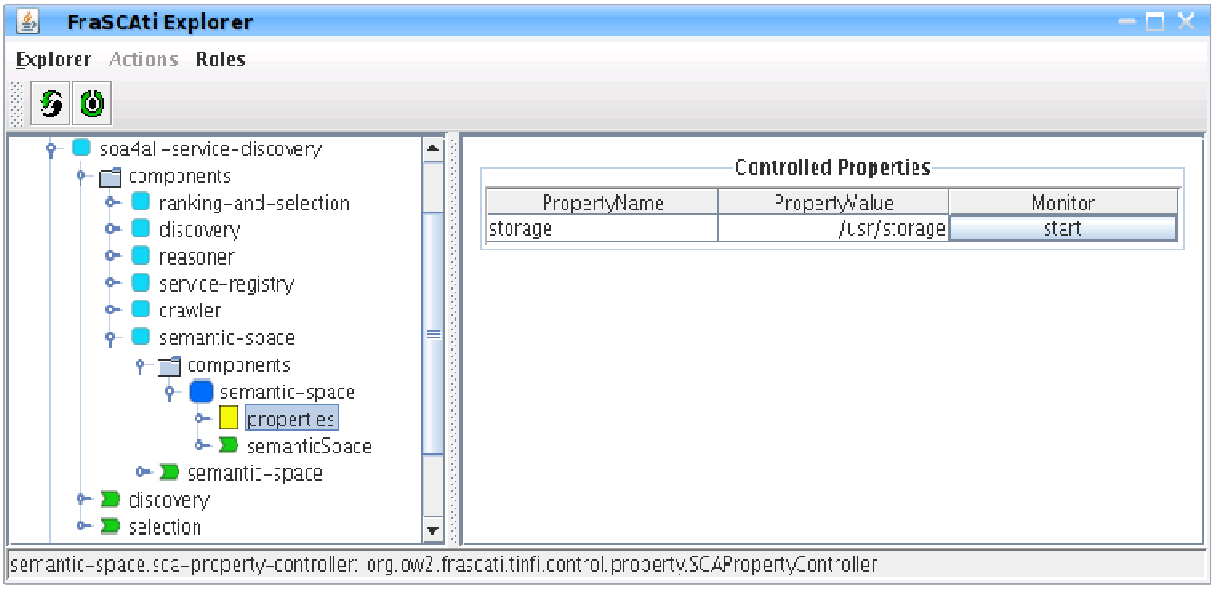


Figure 43: Reconfiguring SCA Properties

5.3.3 Summary

This chapter provided detailed guidelines on how SOA4All Platform Services can be designed, implemented, configured, assembled, tested, managed, and deployed with the SCA standard. We shown that the graphical SCA notation and associated Eclipse STP/SCA Tools can greatly help SOA4All architects to think about the design and integration of SOA4All components (platform services and semantic space). Moreover, SOA4All developers would learn few new rules to implement Java-based SCA components. Yet with SCA, SOA4All architects and developers do not see technical details on the SOA4All DSB implementation, i.e., the DSB transparently transports interactions between SOA4All SCA-based Platform Services. Finally, SCA can bring to SOA4All a continuum from the design to the implementation and execution of the whole SOA4All service-oriented delivery platform.

All the SCA and Java code related to the SOA4All Service Discovery Functional Process is available into the SOA4All SVN at the following URL:

https://svn.sti2.at/SOA4All/trunk/SOA4All-integration/SOA4All-integration-sca

5.4 JBI-Based Platform Services

This chapter provides detailed guidelines on the JBI-based approach to plug SOA4All Platform Services to the SOA4All DSB. It is based on the PEtALS ESB Component Development Kit Framework which allows developers to create compliant JBI components without any JBI specification knowledge.

This chapter is a tutorial for new developers who want to create JBI components. It does not contain any research related work. It just gives concrete details and steps to follow at a quite basic level.

5.4.1 Concrete development steps

The development of Petals components is based on three bricks / tools:

- Apache Maven, and the associated plug-in for Petals.
- An Integrated Development Environment (IDE), for Java development.
- The Component Development Kit (CDK), a Petals library which simplifies the work.

5.4.1.1 Setting up your environment

The first thing to do is to download Apache Maven and an IDE (e.g. Eclipse, NetBeans or IntelliJ).

In the rest of this section, it is assumed you use Eclipse as your IDE. When this document references Eclipse actions, you should be able to find equivalents with other IDEs.

1. Install Maven and make sure that it is in your system path.
2. You should have a variable environment called **M2_HOME** and pointing to the root location of Maven. **M2_HOME/bin** should be present in your system path.

- A mandatory configuration to perform is also related to Maven. It consists in adding the OW2 Maven repositories to your local Maven settings under *$HOME/.m2/seetings.xml*. The settings file is looks like :

```
<settings>
      <profiles>
            <profile>
                  <id>default-profile</id>
                  <activation>
                        <activeByDefault>TRUE</activeByDefault>
                  </activation>
                  <repositories>
                        <repository>
                              <id>ow2-release</id>
                              <name>OW2 Repo</name>
                              <url>http://maven.objectweb.org/maven2
                              </url>
                              <snapshots>
                                    <enabled>false</enabled>
                              </snapshots>
```

```
                        <releases>
                              <enabled>true</enabled>
                        </releases>
                  </repository>
                  <repository>
                        <id>ow2-snapshot</id>
                        <name>OW2 SNAPSHOT Repo</name>
                        <url>http://maven.objectweb.org/maven2-snapshot
                        </url>
                        <snapshots>
                              <enabled>true</enabled>
                        </snapshots>
                        <releases>
                              <enabled>false</enabled>
                        </releases>
                  </repository>
            </repositories>
            <pluginRepositories>
                  <pluginRepository>
                        <id>ow2-plugin</id>
                        <name>OW2 plugin Repo</name>
                        <url>http://maven.objectweb.org/maven2
                        </url>
                        <snapshots>
                              <enabled>false</enabled>
                        </snapshots>
                        <releases>
                              <enabled>true</enabled>
                        </releases>
                  </pluginRepository>
            </pluginRepositories>
      </profile>
  </profiles>
</settings>
```

5.4.1.2 Creating your component

As mentioned several times, JBI components can be Binding Components (BC) or Service Engines (SE). Petals components follow a naming convention:

- *petals-se-componentName* for service engines.

- *petals-bc-componentName* for binding components.

In the scope of this chapter, it is assumed your component is a service engine, ant is called *petals-se-sample*.

To create a new component using the Petals Maven plugin, open a terminal, go into a directory where you can store your project and type the following Maven command line:

```
# Maven command to create a Petals service engine
mvn archetype:create
      -DarchetypeGroupId=org.ow2.petals
      -DarchetypeArtifactId=maven-archetype-petals-jbi-service-engine
      -DarchetypeVersion=1.3.0-SNAPSHOT
      -DgroupId=org.ow2.petals.se.sample
      -DartifactId=petals-se-sample
      -Dversion=1.0-SNAPSHOT
```

The meaning of these options are listed below.

- *archetypeArtifactId is the type of artifact to create.*
  - *maven-archetype-petals-jbi-service-engine for a service engine.*
  - *maven-archetype-petals-jbi-binding -component for a binding component.*
- *archetypeVersion is the archetype version.*
  - *1.3.0-SNAPSHOT is the most recent one at the moment this document is being written.*
- *groupId is the group ID, which is also the name of the root package in the created project.*
- *artifactId is the artifact ID, usually the full component name.*
- *version is the component version, followed by the* -SNAPSHOT *suffix.*

The executed command results in the creation of a Java Maven project.

To transform it into a project we can use in our IDE, we use the following commands:

```
# Go into the created project
cd petals-se-sample
# Convert it into an Eclipse project
mvn eclipse:eclipse
```

You should now wait for Maven to download all the required dependencies and create the Eclipse files. Once this is done, launch your Eclipse IDE and import the project in your workspace: select **File > Import...** and then **General > Existing projects into Workspace**. In the open dialog, click **Browse...**, select your workspace as the root folder and click **OK**. Now, in the listed projects, check your component project and click **OK**.

Your component project is now created and ready to be completed. The main files to notice are:

- The *pom.xml*, which holds the Maven dependencies.
- The *src/main/jbi/jbi.xml*, which is the component JBI descriptor.
- The *src/main/jbi/*.xsd* files, which define the structure of component configurations.
- The *JBIListener.java*, which handles JBI messages in your component.
- The *ServiceEngine* / *BindingComponent* class, which defines the component class.

5.4.1.3 Compiling and deploying your component

Once you have developed your component, the normal next step is to compile it and try to

deploy it on Petals.

To compile your component, open a terminal inside the project directory, and execute one of the following commands:

```
# Just compile
mvn compile
```

```
# Compile and build it for deployment
mvn install
```

With the second command, on a successful build, the result component archive is available in the target folder of your component project. This archive (*.zip) can be deployed in Petals in two ways:

- The first one is to use the Petals administration console.
- The second and most simple one is to copy this archive in the install directory of your Petals installation. This solution perfectly fits development mode.

In any case, you must start Petals before deploying your component.

A successful deployment in Petals is a first validation of your component (but clearly not enough).

5.4.1.4 Testing and debugging your component

5.4.1.4.1 Debugging your component

Although debugging needs you to have some tests to run, it seems a better option to introduce how to debug a component, prior to explain how to test it.

Debugging a component needs three simple steps:

- Add breakpoints in your component.
- Run the Petals debug script (debug.bat / debug.sh).
- In your IDE, use the remote debug feature to launch Petals.

In Eclipse, this is done by selecting **Run > Debug Configurations...**
On the left, right-click on **Remote Java Application** and click **New**.

- In the **Name** field, type in *Petals Remote*.
- In the **Connect** tab, type in the Petals host and port (usually, *localhost* and *8000*).
- In the **Source** tab, make sure your component project is in the displayed sources. If not, click **Add...** In the open dialog, double click **Java Project**, select your project and click **OK**.
- Click **Apply** to save the debug configuration.
- Click **Debug** to launch Petals in debug mode. The terminal in which you launched the debug script should now display that Petals is starting.

You can now build and deploy your component in Petals.
Do not forget to add breakpoints inside your component code. Defining them before or after the build / deployment has no incidence on the debugging.

To make debugging effective, you must manage to make your code to be called.
This is achieved by creating some configurations / tests for your component.

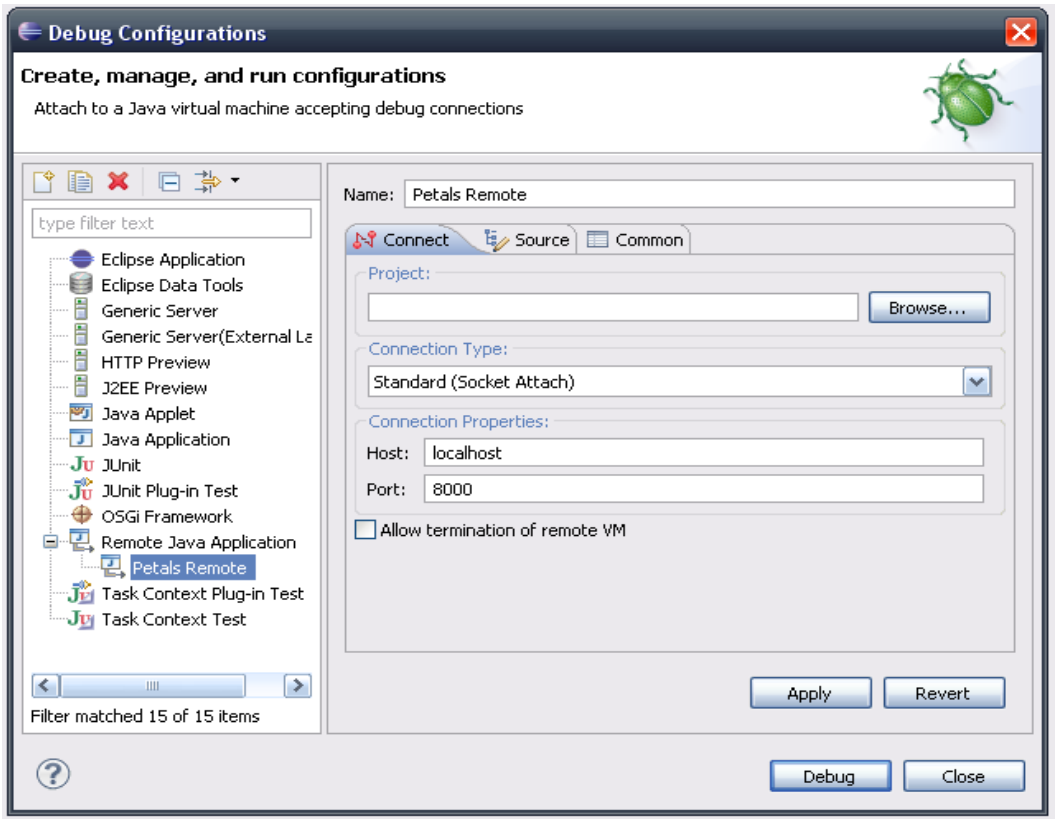


Figure 44 Launching Petals in debug mode from Eclipse

5.4.1.4.2 Testing your component

Testing a component can be seen from two points of view.
The first one is the industrial test campaign, which relies on continuous integration (e.g. Hudson), and which is based on Maven and JUnit. Clearly, it is not in the scope of this document to discuss this kind of tests.

The second way to consider testing is related to debugging, and consists in making small unit tests to check the behavior of your component. They can be seen as check tests that complete code verification. This kind of test requires you to create service assemblies for your component, and play with the different parameters a service-unit can held. Defining such service assemblies is a primary step toward continuous integration, though it is not enough (you will probably not cover all the possible cases).

The methodology for running these tests is the following:

1. Start Petals in debug mode (and add breakpoints if necessary in your component code).
2. Build and deploy your component in Petals.
3. Deploy the service assembly to test with your component (and check that it goes fine).
4. Deploy the sample client component (available on the Petals website and in the Petals Quick Start versions).
5. Once deployed, the sample client launches a Swing dialog which allows you to call any service inside Petals.

In the sample client dialog, go into the **Query** tab and click **Find all the endpoints**.

A list of Petals services appears in the lower part of the dialog. If the deployment of your service assembly went fine, you should find it in the displayed list. Double click it and go back in the **Send** tab. Play with the **Message Exchange patterns** (MEP) and the **In** request (which must be an XML document). Eventually, use the **SendSync** or **Send–Accept** buttons to send your message to your service.

The sent message should be received and processed by your component.
If you added breakpoints, the execution should be suspended when a breakpoint is reached at runtime.

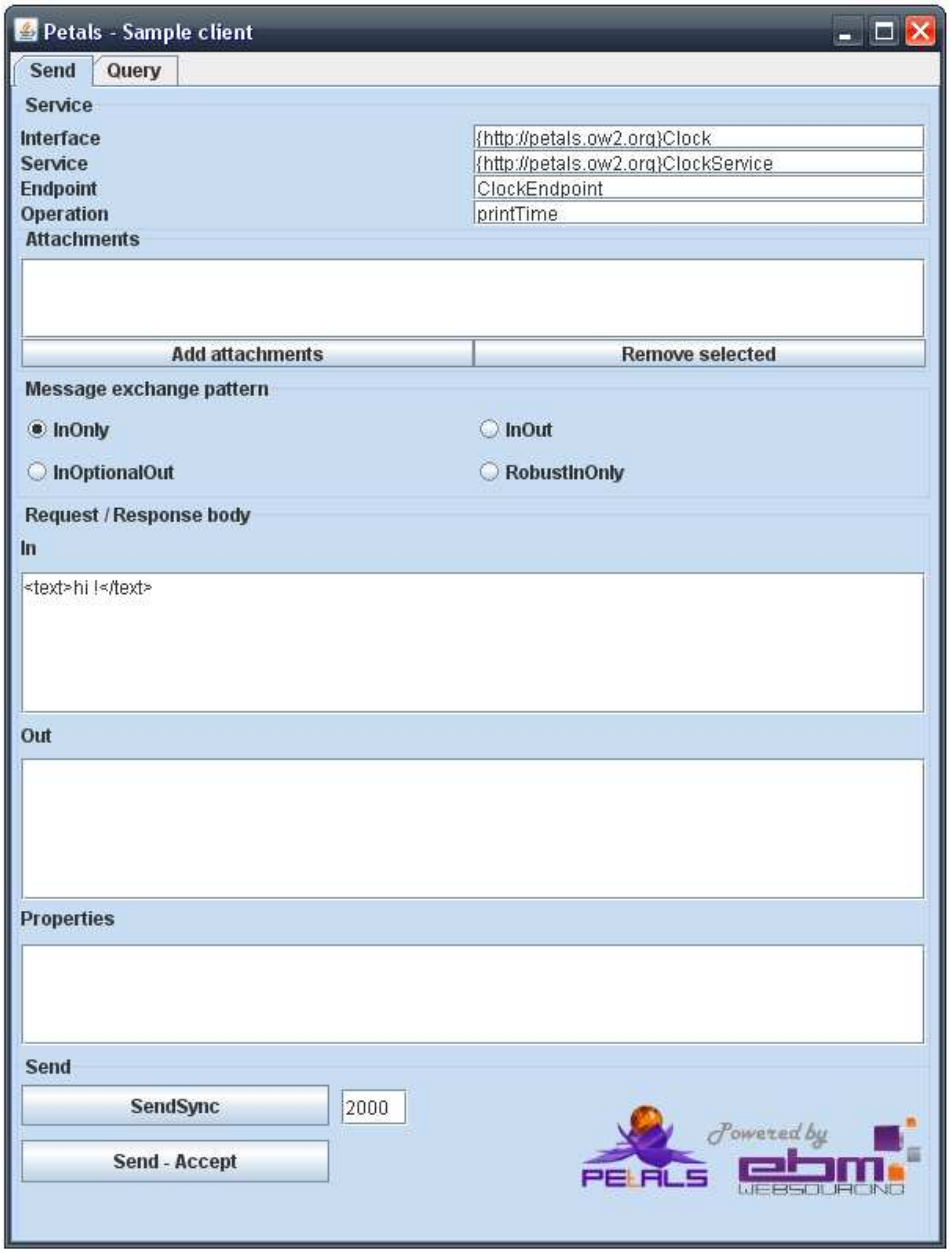


Figure 45 The PEtALS sample client

5.4.2 Introducing and classifying Petals components

The previous part introduced how you could concretely make a Petals component. It did not explain what is a Petals component or what it does. This part introduces these elements and explains the different properties of a Petals component.

5.4.2.1 BC vs. SE

As explained in the introduction, there are two kinds of Petals components.
A Petals component can only be of one kind, not of both.

- Binding components (BC) are links between the services in the bus and external applications. The natural way to provide these bounds is the usage of communications protocols. This is why binding components are generally associated with communication protocols, like SOAP, RMI, JMS...
- Service engines (SE) are pieces of application logics fully integrated in the bus. They are exposed directly as services into the bus. It can be executing code (POJO), performing transformations (XSLT), scheduling (Quartz) and so on...

However, a component by itself does nothing if it is alone. Components are intended to be configured.

5.4.2.2 Component configurations

A configuration for a component is called a service-unit, and is deployed through a service-assembly. These two elements are archive files (*.zip). The service-unit content is used by the component to expose the configuration as a service inside Petals.

A service-unit embeds :

- A JBI descriptor (*META-INF/jbi.xml* file), which defines the provided and consumed services, plus additional parameters.
- A WSDL file.
- Depending on the component task, some other files (e.g. Java code).

The provided WSDL is the interface of the service that will be created by the component from the configuration. Every service in Petals should have a WSDL interface, so that it can be called (e.g. in an orchestration). In the case where a service-unit does not provide a WSDL, and if it makes sense for your component, your component should provide a default WSDL that can be used for the service-unit.

As an example, this what the Mail and the FTP components offer. The components have generic operations like *send()*, *put()*, *get()* and so on. Therefore, they can provide a WSDL by default.

If a service-unit does not embed a WSDL file, and if your component does have a default one, the Petals kernel generates one by default (setting information from the service-unit *jbi.xml* file). But this is definitely a situation to avoid, because the generated WSDL has no operation and consequently, cannot be used in WSDL-driven calls.

The service name, interface name and end-point name are provided in the configuration JBI descriptor. They are used to identify the service in the bus. Every end-point matches a physical service and must have a WSDL interface, as described above.

In the rest of the this document, “service WSDL” are also called “configuration WSDL”, “service-unit WSDL” or “end-point WSDL”. These terms are strictly equivalent (even if the WSDL was provided by default by the component, it then become the service's one).

5.4.2.3 The CDK

In Petals, components interact with the container according to the JBI standard (Java Business Integration). This standard defines interfaces that a component must implement to interact in the JBI environment. To simplify the development of these components, Petals provides an API as a set of classes which implement the required interfaces and makes the work lighter. This library, known as the Component Development Toolkit (CDK), also provides utility methods and a common basis for all the Petals components.

When you go through the steps given in the first part of this document, the built code relies on the CDK.

5.4.2.4 Providers vs. Consumers

In Petals, there are two roles a component can hold. A component may act as a provider, which means it provides a service (receives calls). Or it may act as a consumer, which means it consumes services (sends calls). A component may have both roles at the same time (an orchestration is an example). Component configurations have a *jbi.xml* file in which they define if the component will act as a provider or a consumer. But it is the component implementation which determine whether the component can be (or not) a provider or a consumer.

5.4.2.5 Component's job

A component, in Petals, is responsible for:

- The deployment of a configuration for this component.
- The handling of messages intended for the configurations held by this component.

Hopefully, the deployment part is performed by the CDK (though it can be overridden).

When you create your component, as explained in the first part of this document, the only expected work is to define the behavior of the component when it receives a JBI message. This is done in JBI listener classes.

5.4.2.6 Message dispatching in Petals

One important thing to know before going deeper about how to handle messages in your component, is the way Petals dispatches messages.
For every component based on the CDK, the component JBI descriptor inherits some fields from the CDK.

```
<petalsCDK:acceptor-pool-size>5</petalsCDK:acceptor-pool-size>
<petalsCDK:processor-pool-size>10</petalsCDK:processor-pool-size>
```

- The acceptor pool size defines the number of threads that can accept messages from the bus in concurrency. Once a message is accepted by an acceptor, this message is added to a queue, waiting to by processed by a thread from the processor pool. Since acceptors have few work to do, their number does not have to be high.
- The processor pool is in charge of processing the messages added in the waiting queue. Each thread from the processor pool owns a JBI listener instance. For every message to process, the processor thread initializes the configuration in the JBI listener instance and calls the proper processing method, e.g. *onJBIMessage*. Messages are picked up by the processor pool in FIFO order.

5.4.3 Between theory and practice: handling messages

When you create a new Petals component with Maven, classes are created in a sub-

package (*listener* or *listeners*). The package and class names depend on whether you are working on a binding component or a service-unit. In both cases, the first task to implement your component is the processing of messages, also called exchanges in Petals.

5.4.3.1 Exchanges

An exchange is a JBI message received by the component and addressed to one of its configuration. The component has to use the service configuration (the service-unit content) to process this message.
An exchange holds information about the Message Exchange Pattern (MEP), the called operation (as described in a WSDL file), the input message (an XML document defining the parameters for the called operation), properties, attachments, notifications and the the called service (service, interface, end-point names – these fields allow to determine which configuration use for the processing).
The method body must filter these information to decide which treatment perform.
Typically, check the role or that the called operation supports the MEP, and then start the processing.

The example below is taken for a service engine in the provider role.

```
    @Override
    public boolean onJBIMessage( final Exchange exchange ) {
        try {
            // 'exec' should be called in ROBUST_IN_ONLY
            if( exchange.getOperationName().equals( "exec" )
                    && MEPConstants.ROBUST_IN_ONLY_PATTERN.equals(
exchange.getExchangePattern())) {
                // TODO: execute some code
            }
            else if...
        } catch( MessagingException e1 ) {
            // TODO: create a fault or set an error depending on the MEP
        }
```

Checking MEP and operations is the kind of operation that can rely on a provided WSDL.
But even with a WSDL, dispatching the work according to the called operation cannot be automatic. You have to explicitly check the called operation and decide of the next steps.

5.4.3.2 Message Exchange Patterns

Message exchange patterns, or MEP, are a concept inherited from web services.
They define the way a service is called and the way it should respond. In Petals, there are three used patterns:

IN_ONLY is used for one-way exchanges.
When a component receives an IN_ONLY message, it should set the status of the exchange to DONE and send the response while processing the message (e.g. in a thread). The result of the message processing does not matter for the consumer. The consumer only wants to know whether its message arrived or not. This why the response should be sent as soon as

possible. No fault and no error should be set on the returned exchange with this MEP.

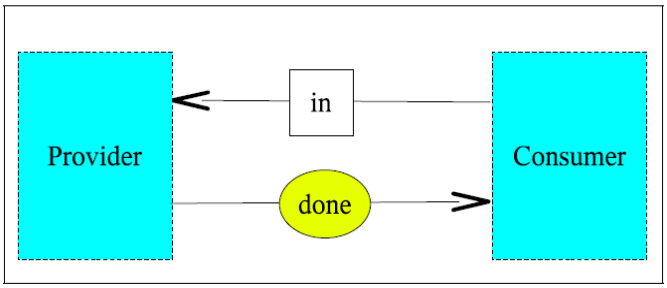


Figure 46 In-Only message exchange pattern

ROBUST_IN_ONLY is used for reliable one-way exchanges.
When a component receives a ROBUST_IN_ONLY message, it can reply by setting the status or by setting a fault on the returned exchange. The decision depends on the processing of the message. If a fault is returned, it should have been declared in the end-point WSDL interface.

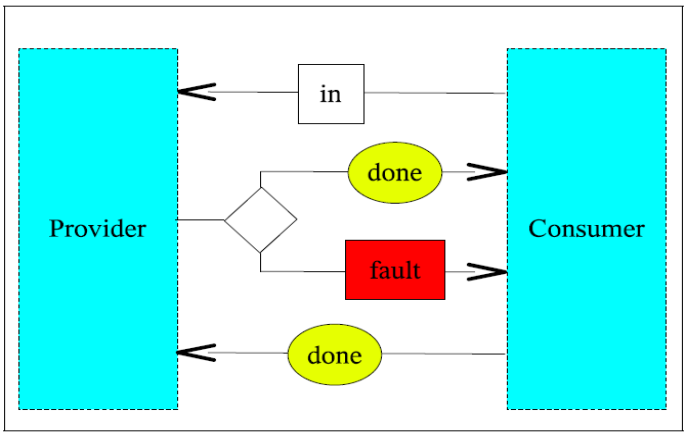


Figure 47 Robust In-Only message exchange pattern

If the provider responds with a status, the exchange is complete.
If it responds with a fault, the consumer responds with a status to complete the exchange.
In this last case, the CDK parameters **ignore-status** determines what is done of this returned status. If it is set to *DONE_AND_ERRORS_IGNORED*, then your JBI listener does not have to process the returned status. The CDK does it for you.

IN_OUT is used for two-way exchanges.
When a component receives an IN_OUT message, it can respond either with a message, or with a fault. The decision depends on the processing of the message. If a fault is returned, it should have been declared in the end-point WSDL interface.

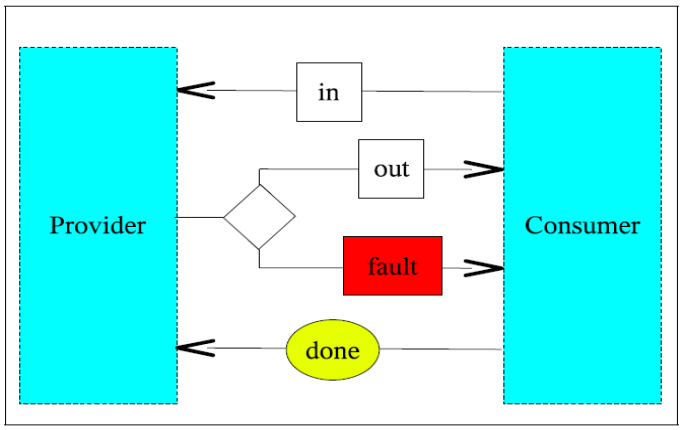


Figure 48 In-Out message exchange pattern

No matter what the provider responds (message or fault), the consumer sends a status to complete the exchange. As in the **ROBUST_IN_ONLY** pattern, the way this status is processed depends on the CDK **ignore-status** parameter.

There is a last MEP, called **IN_OPTIONAL_OUT**, but it is rather complicated and not used in Petals. People interested by this pattern can check the JBI specification, version 1.0.

5.4.3.3 Status, faults, errors and messages

The previous section explained message exchange patterns. But it did not explain concretely how to set a fault or a status on an exchange.

Status are acknowledgments. They can be set by using one of the following Exchange methods:

- Exchange#setActiveStatus(); // Case where you are a consumer
- Exchange#setDoneStatus(); // Case OK
- Exchange#setErrorStatus(); // Case not OK
- Exchange#setStatus(ExchangeStatus status);

Faults express a business fault (with respect to the service logic). They can be set by using one of the following methods:

- Exchange#setFault(Throwable e);
- Exchange#setFault(Fault fault);

Errors express errors at the messaging level.They can be set by using:

- Exchange#setError(Exception exceptionError);

Messages express a normal result (with respect to the service logic). They can be set in the returned exchange by using:

- Exchange#setOutMessage(NomalizedMessage msg);
- Exchange#setOutMessageContent(Document outContent);
- Exchange#setOutMessageContent(Source outContent);
- Exchange#setOutMessageContent(String outContent);

Building the out content has to be made in the component. The out content structure should respect the WSDL out message schema of the called operation.

5.4.3.4 Processing exchanges in Petals components

Obviously, the code to execute in the component depends on its job. The dispatching has to be done in the JBI listener. The rest of the processing can be written inside the method body, or delegated to another class (which is a good solution if you have to manage resources or a separate pool). The best thing to do to get examples of processing or patterns, is to check the Petals forge, and take a look at existing components and how they are implemented. One thing maybe to highlight, is that a JBI listener has access to the component instance. The component instance can be used to get a logger. It can also be extended and used to store objects (e.g. a class instance managing resources).

```
      getComponent().getLogger().severe(
                  "Received a call for unknown operation '" +
exchange.getOperationName() + "'." );
```

5.4.3.5 Processing exchanges as a provider

Components acting as a provider expose services from configurations in **provide** mode. That is to say the service-unit *jbi.xml* has a *provides* mark-up.

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi:jbi version="1.0"
      xmlns:jbi="http://java.sun.com/xml/ns/jbi"
      xmlns:petalsCDK="http://petals.ow2.org/components/extensions/version-5"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <jbi:services binding-component="false">
            <jbi:provides
                  interface-name=...
```

As providers expose services, they should be implemented to respond to calls. It means they receive messages for a determined operation, check the operation, the MEP, the role, and then dispatches the work depending on the called operation.

In terms of message processing, a provider must:

- Process calling messages.
- Set a status, a fault or an out message on the returned exchange (depends on the MEP).
- Optionally, take acknowledgments (status) into account (depends on the MEP). This last part can be configured in the component *jbi.xml*, so that the CDK handles this instead of your component. See section 5.1.a for more details about the CDK parameters.

The provider role must respect the message exchange pattern of the calling message. This is illustrated in the section 3.2 about MEP (the diagrams show the provider role).

5.4.3.6 Processing exchanges as a consumer

Components acting as a consumer calls services. The services to call are defined in configurations being in **consume** mode. That is to say the service-unit *jbi.xml* has a *consumes* mark-up.

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi:jbi version="1.0"
      xmlns:jbi="http://java.sun.com/xml/ns/jbi"
      xmlns:petalsCDK="http://petals.ow2.org/components/extensions/version-5"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <jbi:services binding-component="false">
            <jbi:consumes
                  interface-name=...
```

As consumers call services, they should be implemented to send calls and also to process responses to these calls. It means they send messages for a determined operation, with a given MEP and parameters for the operation. The caller also decides whether the call is asynchronous or synchronous. In any case, the send messages should respect the WSDL interface of the called service. The messages sent by a consumer component will be received and processed by a provider component. In terms of message processing, a consumer must:

- Build and send messages to services.
- Wait for responses and process them according to the MEP. If the response exchange was set with a status, the exchange is terminated. Otherwise, the consumer has to send back an acknowledgment (status) to the provider and thus close the exchange. All of this is specified in the section 3.2 about MEP (the diagrams show the consumer role).

Processing the responses can also permit to implement retry policies.
In any case, the consumer role must respect the WSDL interface of called services (message exchange pattern, operation names, operation parameters, returned result).

5.4.3.7 JBI listeners in Service Engines

In service engines, the Petals Maven plug-in creates a *listener* package and two classes: *ServiceEngine.java* (which should be renamed) and *JBIListener.java*.
The first class is mainly used to manage the component life cycle or to store data for JBI listeners. It is discussed in the life cycles section.
The last class extends the abstract class *org.ow2.petals.component.framework.listener.AbstractJBIListener* .
It requires one method to be completed:

```
public abstract boolean onJBIMessage( Exchange exchange );
```

It is in this method that the provider and/or the consumer role are implemented.
The result of the onJBIMessage method needs some clarifications with respect to its Java doc. Indeed, if the processing when the message arrives is defined by the component, the returned message can be completed and sent by the component or by the CDK. Let's take our previous example (SE in provider role) and let's complete it in two different ways.

```
    @Override
    public boolean onJBIMessage( final Exchange exchange ) {
```

```
        try {
            // 'exec' should be called in ROBUST_IN_ONLY
            if( exchange.getOperationName().equals( "exec" )
                    && MEPConstants.ROBUST_IN_ONLY_PATTERN.equals(
exchange.getExchangePattern())) {
                // TODO: execute some code
            }
        } catch( MessagingException e1 ) {
            // TODO: create a fault or set an error depending on the MEP
        }
        return true;
    }
```

By returning true, we let the CDK manage the call back message.

```
    @Override
    public boolean onJBIMessage( final Exchange exchange ) {
        try {
            // 'exec' should be called in ROBUST_IN_ONLY
            if( exchange.getOperationName().equals( "exec" )
                    && MEPConstants.ROBUST_IN_ONLY_PATTERN.equals(
exchange.getExchangePattern())) {
                Runnable runnable = new Runnable() {
                    public void run() {
                        // TODO: execute some code
                        // bla-bla

                        // Update the exchange, depending on the MEP, e.g.
                        exchange.setDoneStatus();

                        // Send the response
                        send( exchange );
                    }
                };
                Thread myThread = new Thread( runnable );
                myThread.start();
            }
        } catch( MessagingException e1 ) {
            // TODO: create a fault or set an error depending on the MEP
```

```
        }
        return false;
    }
```

By returning false, it explicitly indicates to the CDK that the component will send back the exchange to the client (the one which called this service, also known as a consumer).
This second solution is best suited for asynchronism calls. As an example, you could imagine putting the runnable into a blocking queue and process it later. In this case, we would use the component (or any other unique instance) to hold this queue.
However, since JBI listeners are already into a pool of threads and associated with a waiting queue, the use of another pool is not a best practice.

5.4.3.8 JBI listeners in Binding Components

In binding components, the Petals Maven plug-in creates a *listeners* package and three classes: *BindingComponent.java* (which should be renamed), *JBIListener.java* and *ExternalJBIListener.java*.
The first class is mainly used to manage the component life cycle or to store data for JBI listeners. It is discussed in the life cycles section. The *JBIListener* class is the same than in service engines. It is used to process messages received from Petals services, that is to say from services inside the bus. The *ExternalListener* class is used to process messages from service outside the bus. Since binding components are connectors between services inside the bus and external applications / services, they must be able to listener calls from services inside the bus (*JBIListener*) and from outside the bus (*ExternalListener*). It extends the abstract class *org.ow2.petals.component.framework.listener.AbstractExternalListener* .
It requires two methods to be completed:

```
public abstract boolean start() throws PEtALSCDKException
public abstract boolean stop() throws PEtALSCDKException
```

These two methods are called when the binding component is started and stopped. They should start and stop a listening service to implement in this class. As an example, the FileTransfer binding component starts polling a directory when the *start()* method is called, and stops polling when *stop()* is called. For network related binding components, it can be creating a socket to listen calls.Examples of binding components are available in the Petals source forge. They can be used as a complement of information.

5.4.4 Between theory and practice: managing life cycles

5.4.4.1 Component life cycle

Every artifact in Petals has a life cycle. Components do not break the rule. The diagram below illustrates the different states of a component.

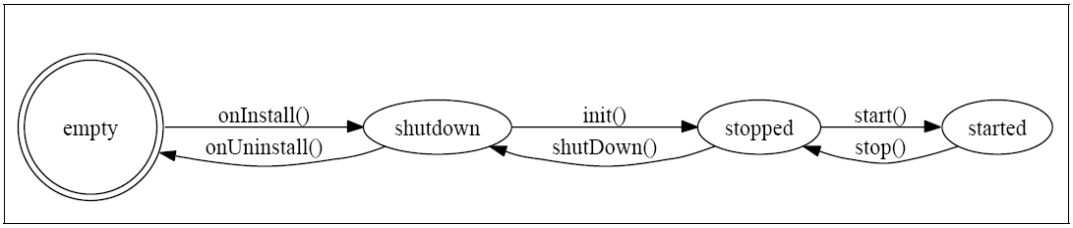


Figure 49 Life cycle of a Petals component

5.4.4.2 Component parameters

As explained previously in this document, a component has a JBI descriptor which specifies parameters. Among these parameters, there are standard parameters (from in the JBI specification), CDK parameters (explained farther) and component specific parameters. Component specific parameters are parameters which configure your component behavior. To define a new component parameter, simply add a new element in your *jbi.xml* file.
The example below defines an new element, called executor-*size*.

jbi.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi:jbi xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:petalsCDK="http://petals.ow2.org/components/extensions/version-5"
    xmlns:jbi="http://java.sun.com/xml/ns/jbi" version="1.0"
    xmlns:petals-se-sample="http://petals.ow2.org/components/petals-se-sample/version-1.0-SNAPSHOT">

    <jbi:component type="service-engine">
        <jbi:identification>
            <jbi:name>petals-se-sample</jbi:name>
            <jbi:description>petals-se-sample service engine</jbi:description>
        </jbi:identification>
        <jbi:component-class-name>org.ow2.petals.se.sample.ServiceEngine</jbi:component-class-name>
        <jbi:component-class-path><jbi:path-element/></jbi:component-class-path>
        <jbi:bootstrap-class-name>org.ow2.petals.component.framework.DefaultBootstrap</jbi:bootstrap-class-name>
        <jbi:bootstrap-class-path><jbi:path-element/></jbi:bootstrap-class-path>

        <!-- CDK specific fields -->
        <petalsCDK:acceptor-pool-size>5</petalsCDK:acceptor-pool-size>
        <petalsCDK:processor-pool-size>10</petalsCDK:processor-pool-size>
        <petalsCDK:ignored-status>DONE_AND_ERROR_IGNORED</petalsCDK:ignored-status>
        <petalsCDK:jbi-listener-class-name>org.ow2.petals.se.sample.listeners.JBIListener</petalsCDK:jbi-listene

        <!-- Component specific configuration -->
        <petals-se-sample:executor-size>5</petals-se-sample:executor-size>

    </jbi:component>
</jbi:jbi>
```

Figure 50 Adding a component parameter in the jbi.xml

Notice that this element is associated to the component name space. All the component parameters must be associated to the component name space. To see how to retrieve this parameter in your component, see the sample given in the next section.

5.4.4.3 The component deployment

When you created your component with Maven, you must have got a class named ServiceEngine or BindingComponent. This class is generated by the Maven plug-in for commodity reasons. It allows you to find it easily.
The effective component class is the one specified in the component JBI descriptor (*src/main/jbi/jbi.xml*), in the **component-class-name** element. If you have nothing to do in this class, you can delete it and set a default component class in your *jbi.xml*:

- org.ow2.petals.component.framework.se.DefaultServiceEngine
- org.ow2.petals.component.framework.bc.DefaultBindingComponent

It will manage the component life cycle and configurations for you. Notice that unlike JBI listeners, there is only one component class instance per component name. The pool of JBI listeners is managed by the component instance. If you have specific actions to perform on key events in the component life cycle, you need to have a custom component class. In this case, the important component methods are:

```
protected void doInit() throws JBIException {}
```

Is called when the component is deployed (from the shutdown to the stopped state).

```
protected void doStart() throws JBIException {}
```

Is called when the component is started.

```
protected void doStop() throws JBIException {}
```

Is called when the component is stopped.

```
protected void doShutdown() throws JBIException {}
```

Is called when the component is shutdown.

Here is an example which creates an executor service when our component is deployed. When the component stops, the executor stops accepting new threads. When the component is shutdown, threads which were still processed are stopped.

```
public class ServiceEngine extends AbstractServiceEngine {
    private static int DEFAULT_POOL_SIZE = 5;
    private ExecutorService executor;
    /**
     * Initializes the executor service from the component configuration.
     * @see org.ow2.petals.component.framework.AbstractComponent#doInit()
     */
    @Override
    protected void doInit() throws JBIException {
        // Get the executor size from the component configuration
        int poolSize = DEFAULT_POOL_SIZE;
        for( Element elt : getComponentConfiguration().getAny()) {
            if( "executor-size".equals( elt.getLocalName().toLowerCase())) {
                try {
                    poolSize = Integer.valueOf( elt.getTextContent().trim());
                    if( poolSize < 1 )
                        poolSize = DEFAULT_POOL_SIZE;
                } catch( Exception e ) {
                    getLogger().info( ''Could not retrieve the pool size from
the JBI descriptor.'' );
                }
                break;
            }
        }
        // Create the executor
        executor = Executors.newFixedThreadPool( poolSize );
    }

    /**
     * Do not accept any other thread.
```

```
     * @see org.ow2.petals.component.framework.AbstractComponent#doStop()
     */
    @Override
    protected void doStop() throws JBIException {
        executor.shutdown();
    }

    /**
     * Stop processing the threads.
     * @see org.ow2.petals.component.framework.AbstractComponent#doShutdown()
     */
    @Override
    protected void doShutdown() throws JBIException {
        executor.shutdownNow();
    }

    /**
     * Delegate method.
     * @param <T>
     * @param task
     * @return a future
     * @see
java.util.concurrent.ExecutorService#submit(java.util.concurrent.Callable)
     */
    public <T> Future<T> submit( Callable<T> task ) {
        return executor.submit( task );
    }
}
```

5.4.4.4 Deployment of a configuration

Component configurations (service units) are used by the component to create services in the bus. Remember that these services should have a WSDL interface, either provided by in the service-unit or by the component. When a message is sent to one of these services, it is received by the component which then uses the associated configuration to perform the treatment. Configurations parameters and resources can be used at several moments:

- At deployment: the configuration influences the way resources are created.
- At runtime: the configuration is used to determine how to process a message.
- At any other life cycle moment, e.g. to determine what to do when the service stops.

If configurations are used beyond the deployment phase, they should be cached at deployment. One way to do that is to build a structure maintained by the component. Then,

JBI listeners will access the component data and determine how to process a message depending on the end-point relative data. This is achieved with a SU manager. A SU manager is responsible for the life cycle of a service-unit (component configuration)

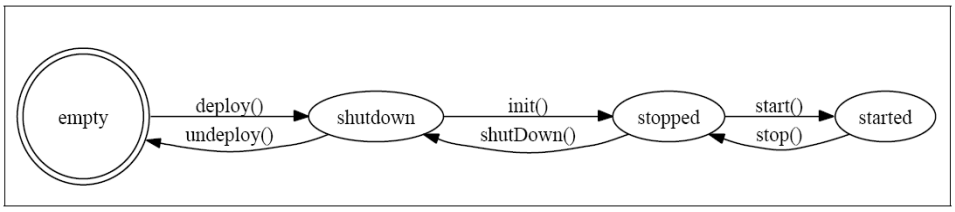


Figure 51 Life cycle of a service-unit

The life-cycle of a service-unit is very similar to the life cycle of a component. Therefore, a SU manager has similar methods to a component (init / start / stop / shutdown).
In the example below, the SU manager overrides the deployment of a configuration to store parameters retrieved from the configuration *jbi.xml* file (the *jbi.xml* file in the service-unit). To ease the comprehension, a sample service-unit jbi.xml file is also given.

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi:jbi version="1.0"
      xmlns:jbi="http://java.sun.com/xml/ns/jbi"
      xmlns:petalsCDK="http://petals.ow2.org/components/extensions/version-5"
      xmlns:myComponent="http://petals.ow2.org/components/myComponent/version-
1.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <jbi:services binding-component="false">
            <jbi:provides
                  interface-name="genNs:myServiceItf"
                  service-name="genNs:myServiceName"
                  endpoint-name="myServiceEndpoint"
                  xmlns:genNs="http://ebmwebsourcing.com/myService">
                  <!-- CDK parameters -->
                  <petalsCDK:wsdl>myWsdl.wsdl</petalsCDK:wsdl>
                  <!-- Component specific parameters -->
                  <myComponent:param1>myParam1</myComponent:param1>
                  <myComponent:param2>myParam2</myComponent:param2>
            </jbi:provides>
      </jbi:services>
</jbi:jbi>
```

In this example, the component specific parameters are two elements called *param1* and *param2*. This is how your component can get and store them for a later use.

```
public class SampleSeSUManager
extends ServiceEngineServiceUnitManager
implements ServiceUnitManager {
    /**
     * Constructor.
     * @param engine
     */
    public SampleSeSUManager( ServiceEngine engine ) {
        super( engine );
    }

    /**
     * Handles the deployment of a service-unit.
     * @see org.ow2.petals.component.framework.su.AbstractServiceUnitManager
     * #deploy(java.lang.String, java.lang.String)
     */
    @Override
    public String deploy( String serviceUnitName, String suRootPath ) throws
DeploymentException {
        String deploymentResult = super.deploy( serviceUnitName, suRootPath );

        // Cache data related to the job
        ServiceUnitDataHandler suDataHandler =
getServiceUnitDataHandlers().get( serviceUnitName );
        List<Provides> providesList =
suDataHandler.getDescriptor().getServices().getProvides();

        if( providesList == null || providesList.size() != 1 )
            throw new DeploymentException( "One and only one provides has to be
declared" );

        Provides provides = providesList.get( 0 );
        ConfigurationExtensions extensions =
suDataHandler.getConfigurationExtensions( provides );
        try {
            // Cache configuration elements
            Map<String,String> endpointNameToSuParam = new
HashMap<String,String>();
            if( extensions != null ) {
```

```
                // Getting simple elements, called "param1" and "param2"
                String suParam1 = extensions.get( "param1" );
                // TODO: store the param1 in the component
                String suParam2 = extensions.get( "param2" );
                // TODO: store the param2 in the component
            }
        } catch (Exception e) {
            throw new DeploymentException( "Caching SU data failed." );
        }
        return deploymentResult;
    }
}
```

Let's now suppose your service-unit jbi.xml has several elements with the same name, that is to say a list of elements. The component specific part in the service-unit would look like:

```
<!-- Component specific parameters -->
<myComponent:paramName>myValue1</myComponent:paramName>
<myComponent:paramName>myValue2</myComponent:paramName>
<myComponent:paramName>myValue3</myComponent:paramName>
<myComponent:param2>myParam2</myComponent:param2>
```

Here, there are two kinds of elements: *paramName* and *param2*. To retrieve the list of elements, Petals has an unsophisticated (or inconvenient, you decide) way to proceed. Indeed, when they are parsed and stored in the configuration, the list elements are renamed. Here, it would result in the following element list: [*paramName1, paramName2, paramName3, param2*]. The code above would be a little bit modified:

```
public class SampleSeSUManager
extends ServiceEngineServiceUnitManager
implements ServiceUnitManager {
    /**
     * Constructor.
     * @param engine
     */
    public SampleSeSUManager( ServiceEngine engine ) {
        super( engine );
    }

    /**
```

```
     * Handles the deployment of a service-unit.
     * @see org.ow2.petals.component.framework.su.AbstractServiceUnitManager
     * #deploy(java.lang.String, java.lang.String)
     */
    @Override
    public String deploy( String serviceUnitName, String suRootPath ) throws
DeploymentException {
        String deploymentResult = super.deploy( serviceUnitName, suRootPath );
        // Cache data related to the job
        ServiceUnitDataHandler suDataHandler =
getServiceUnitDataHandlers().get( serviceUnitName );
        List<Provides> providesList =
suDataHandler.getDescriptor().getServices().getProvides();


        if( providesList == null || providesList.size() != 1 )
            throw new DeploymentException( "One and only one provides has to be
declared" );


        Provides provides = providesList.get( 0 );
        ConfigurationExtensions extensions =
suDataHandler.getConfigurationExtensions( provides );
        try {
            // Cache configuration elements
            Map<String,String> endpointNameToSuParam = new
HashMap<String,String>();
            if( extensions != null ) {
                List<String> paramNames = new ArrayList<String>();
                for( Map.Entry<String,String> entry : extensions.entrySet()) {
                        if( entry.getKey.startsWith( "paramName" ))
                                paramNames.add( entry.getValue());


                         if( entry.getKey.equals( "param2" ) {
                                // TODO: store the param2 in the component
                        }
                }
                // TODO: store the list of paramNames in the component
            }
        } catch (Exception e) {
            throw new DeploymentException( "Caching SU data failed." );
        }
```

```
        return deploymentResult;
    }
}
```

Eventually, we would have to make sure this SU manager will be used instead of the component default one. This is achieved by changing one method in the component class.

```
    /*
     * (non-Javadoc)
     * @see org.ow2.petals.component.framework.se.AbstractServiceEngine
     * #createServiceUnitManager()
     */
    @Override
    protected AbstractServiceUnitManager createServiceUnitManager() {
        return new SampleSeSUManager( this );
    }
```

As a general principle, the component should store service-unit data in a Map, where the key is the end-point name, and the value an object embedding the service-unit data. This map is populated by the SU Manager instance. When a JBI listener receives an exchange, it gets the target end-point, retrieves the associated data through the component, and then decides of the processing.

5.4.5 Summary

This section explained all the steps to follow to create a JBI component from scratch using the PEtALS ESB Tools such as the Component Development Kit, Maven plugin and Eclipse IDE. It also introduced some JBI aspects which gives more understanding of the JBI concept and philosophy.

5.5 List of Platform Services

This chapter provides the list of available SOA4All platform services.

5.5.1 Space Service

The Space Service is available as a:

- JBI Endpoint at {http://ws.space.dsb.SOA4All.eu/}SemanticSpaceWSImplService
- Web service at http://<HOST>/petals/services/SpaceService

Note: At the implementation level, the Web service and the JBI service are the same runtimes, the JBI service is exposed as Web service with the help of the PEtALS ESB SOAP Binding Component.

6. Conclusions

This deliverable provides a written report to the first SOA4All Distributed Service Bus implementation. The implementation delivers the software that is necessary to enable large-scale installations of service bus nodes. These nodes provide the backbone of the SOA4All Runtime and host the core infrastructural services of SOA4All. As specified in in Deliverable D1.4.1A, the Distributed Service Bus serves as integration platform for SOA4All Platform Services, as well as 3rd party business services. While the semantic space implementation, as core element of the bus, i presented in Deliverable D1.3.2B, the software descriptions in this deliverable focus on the technical registry, the messaging infrastructure and the monitoring platform. As a whole, this first bus prototype implementation is realized and deployable on top of ProActive.

As stated above, the main objective of the deliverable was to present the prototype implementation. As such, the document provides detailed descriptions of the different components of the prototype in order to better present the work done and the functionality provided. In addition, there are a section with installation and configuration guidelines and one the present in more detail theintegration interfaces and the monitoring and management APIs. Last but not least, a part of this deliverable is dedicated to integration guidelines for developers of services about how to publish services to the bus, or how to host services on the bus as SCA or JBI components.

The work on component (i.e., platform service) integration will be continued in the period toward month M24, and the realization of functional processes, as compositions of platform services, will be one of the main efforts of that period. The results of these investigations will be one of the main subjects to deliverable D1.4.2A "Final SOA4All Reference Architecture Specification".

7. References

1. E. Mathias, V. Cavé, F. Baude. A GCM-Based Runtime Support for Parallel Grid Application. In: Proc. of the ACM SIGPLAN Component-Based High-Performance Computing (CBHPC) 2008, Karlsruhe, Germany.
2. Reto Krummenacher, SOA4All Architecture, slides for the 2nd SOA4All Review Meeting, Brussels, Belgium, April 2, 2009. https://portal.atosorigin.es/pls/portal30/docs/FOLDER/SOA4ALL_PROJECT_AREA/MEETINGFOLDERS/9YEARLYPROJECTREVIEWBRUSSELS23APRIL2009/PRESENTATIONS/WP1-SOA4ALL-ARCHITECTURE-RKR.PDF
3. Reto Krummenacher (UIBK), Imen Filali (INRIA), Fabrice Huet (INRIA) Françoise Baude (INRIA), Distributed Semantic Spaces: A Scalable Approach to Coordination, Deliverable 1.3.2A, SOA4All Consortium, March 2009. http://www.SOA4All.eu/docs/D1.3.2A%20Distributed%20Semantic%20Spaces%20A%20Scalable%20Approach%20To%20Coordination.pdf
4. Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani (INRIA), Reconfigurable SCA Applications with the FraSCAti Platform, In Proceeding of the 6th International Conference on Services Computing (IEEE SCC 2009), Bangalore, India, September 21-25, 2009. To appear. http://hal.inria.fr/inria-00397856
5. OASIS Open. Web Services Topics (WS-Topics). OASIS Open, Version 1.3. October, 2006.
6. C. Pedrinaci, J. Domingue, A. Alves de Medeiros. A Core Ontology for Business Process Analysis. In Proceeding of 5th European Semantic Web Conference (ESWC 2008), 2008, Tenerife, Spain
7. Elmo, Version 1.5. http://www.openrdf.org/doc/elmo/1.5/
8. Drools: Business Logic integration Platform. http://www.jboss.org/drools/
9. W3C, SPARQL Query Language for RDF, January 15 2008. http://www.w3.org/TR/rdf-sparql-query/
10. E. Mathias, V. Cav´e, S. Lanteri, and F. Baude. Grid-enabling SPMD Applications through Hierarchical Partitioning and a Component-Based Runtime. In 15th Int. European Conf. on Parallel and Distributed Computing (Euro-Par 2009), August 2009. To appear. Reviewers can access the camera-ready www.inria.fr/oasis/Francoise.Baude/europar09.pdf
11. R. Krummenacher, I. Toma, Ch. Hamerling, J.-P. Lorre, F. Baude, V. Legrand, Ph. Merle, C. Ruz, C. Pedrinaci, D. Liu, and T. Pariente Lobo: SOA4All Reference Architecture Specification. SOA4All project deliverable D1.4.1A, March 2009. http://www.SOA4All.eu/docs/D1.4.1A%20SOA4All%20Reference%20Architecture.pdf
12. R. Krummenacher, M. Fried, F. Huet, I. Filali, L. Pellegrino, and Ch. Hamerling: Distributed Semantic Spaces: A First Implementation. SOA4All project deliverable D1.3.2B, August 2009.
13. Sun Microsystems, Inc., The Java API for XML-Based Web Services (JAX-WS) 2.0, Final Release, JSR-000224 Specification, April 19, 2006. http://jcp.org/aboutJava/communityprocess/final/jsr224/index.html

14. Open SOA, SCA Service Component Architecture – Assembly Model Specification, SCA Version 1.00, March 15, 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

15. Open SOA, SCA Service Component Architecture – Java Common Annotations and APIs, SCA Version 1.00, March 21, 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications