Petals ESB explained

The Open Source Entreprise Service Bus solution for Service Oriented Architectures

Version history

Version	Date	Comments, Changes, Status	Authors, Contributors, Reviewers
1.0	2009-01-11		Christophe HAMERLING – eBM WebSourcing

Table of Contents

Figures

Acronyms

Please find below the acronyms used in this document and their definition.

Acronym	Definition
API	Application Programming Interface
BC	Binding Component
CBD	Component Based Development
EDA	Event Driven Architecture
ESB	Entreprise Service Bus
JBI	Java Business Integration
JCP	Java Community Process
JMS	Java Messaging Service
JNDI	Java Naming and Directory interface
JSR	Java Specification Request
LDAP	Lightweight Directory Access Protocol
NMR	Normalized Message Router
SA	Service Assembly
SE	Service Engine
SL	Shared library
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SPI	Service Provider Interface
SU	Service Unit
WS	Web Service
WSDL	Web Service Description Language
WSDM	Web Service Distributed Management

Introduction to Services, Web Services and SOA

Service

OASIS (Advancing Open Standard for the Information Society http://www.oasis-open.org) defines a service as a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. A service is provided by an entity (the service provider) for use by others, but the eventual consumers of the service may not be known to the service provider and may demonstrate uses of the service beyond the scope originally conceived by the provider.

A service is accessed by means of a service interface, where the interface comprises the specifics of how to access the underlying capabilities. There are no constraints on what constitutes the underlying capability or how access is implemented by the service provider.

Thus, the service could carry out its described functionality through one or more and/or manual processes that themselves could invoke other available services.

Web Service

The current Web is mainly a collection of information but does not yet provide support in processing this information, i.e., in using the computer as a computational device. Recent efforts around UDDI, WSDL, and SOAP lift the Web to a new level.

Software applications can now be accessed and executed via the Web based on the idea of Web Services. Web Services significantly increase the Web architecture's potential, by providing a way to automate the communication between distributed applications and the discovery or execution of remote services.

Web Services connect computers and devices using the Internet to exchange data and to combine data and processes in new ways. Web Services can be completely decentralized and distributed over the Internet and accessed by a wide variety of communication devices.

Note: More about services can be found in the Reference Model for Service Oriented Architecture OASIS Standard (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm).

Service Oriented Architecture

Definition

Service Oriented Architecture (SOA) is a computer system's architectural style for creating and using business processes, packaged as services, throughout their lifecycle. SOA also defines and provisions the IT infrastructure to allow different applications to exchange data and participate in business processes. These functions are loosely coupled with the operating systems and programming languages underlying the applications. SOA separates functions into distinct units (services), which can be distributed over a network and can be combined and reused to create business applications. These services communicate with each other by passing data from one service to another, or by coordinating an activity between two or more services. SOA concepts are often seen as built upon, and evolving from older concepts of distributed

computing and modular programming.

The following guiding principles define the ground rules for development, maintenance, and usage of the SOA:

- Reusability, granularity, modularity, composability, componentization and interoperability
- Compliance to standards (both common and industry-specific)
- Services identification and categorization, provisioning and delivery, monitoring and tracking

Infrastructure

A vision of the standard SOA infrastructure is given is the following figure:

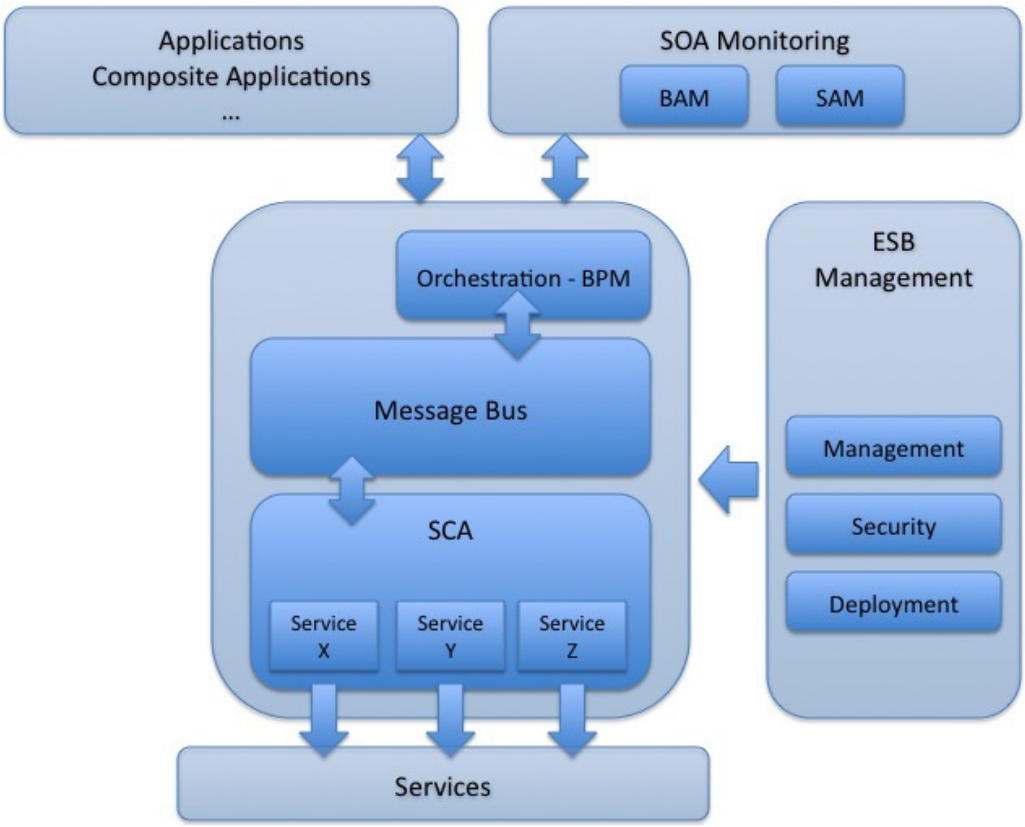

Figure 1 - SOA Infrastructure

This infrastructure is composed of the following modules:

- A message delivery bus, also named the Entreprise Service Bus (ESB)
- A service orchestration engine (BAM - Business Process Management)
- A service container (such as SCA – Service Component Architecture)
- Monitoring Tools (BAM – Business Activity Management, SAM – Service Activity Monitoring)

- Management Tools
- …

The Entreprise Service Bus

Definition

The key item for integration of services within an SOA is the ESB. The goal of an ESB is to provide virtualization of the enterprise resources, allowing the business logic of the enterprise to be developed and managed independently of the infrastructure, network, and provision of those business services.

An ESB must provide a standard and flexible access to the services to the application developer. The service consumer must be as independant as possible from:

- The communication protocol between service consumer and provider. The service consumer must be able to access to the service via HTTP but also via FTP, JMS, JCA, SMTP, RMI, etc…
- The service deployment technology. A Web Service, .NET component, an EJB or a simple Java class may be accessed by the service consumer in the same way.
- The service localisation.

The ESB may also allow the service developer to deploy its services whatever the technology or the protocol he chose to develop the service.

Ideally, the bus must:

- Support the Web Service Description Language – WSDL (WSDL 1.1 http://www.w3.org/TR/wsdl and WSDL 2.0 http://www.w3.org/TR/wsdl20/) as the universal service description language. This is true even if the service is not a Web Service (an EJB for example).
- Interact with clients and services through heterogenous protocols:
    - Not WS-* based (e.g. HTTP, JMS, .NET, …)
    - WS-* based (e.g. SOAP, WS-RM, WS-Adressing, …)
- Provide several modes to call services (synchrone, asynchrone, …)
- Trace the service calls
- Secure message exchanges
- Host local services (SCA)
- Provide management facilities
- …

Most of the ESB are based on a asynchronous messages approach. It means that, whatever the protocol used from the consumer to the ESB and from the ESB to the

service, the ESB internally transform these calls to messages. Next, it insures the routing and transmission of the message to the service in a transparent way from the service consumer and the service provider. The routin gis generally based on the message header (cf WS-Adressing), but a more advanced routing can be based on the message content (message payload). Furthermore, the routing phase can be done with or without acknowledgement, with or without delivery waranty, …

Architecture

The following figure introduces the functional modules that are needed to satisfy the previous objectives.

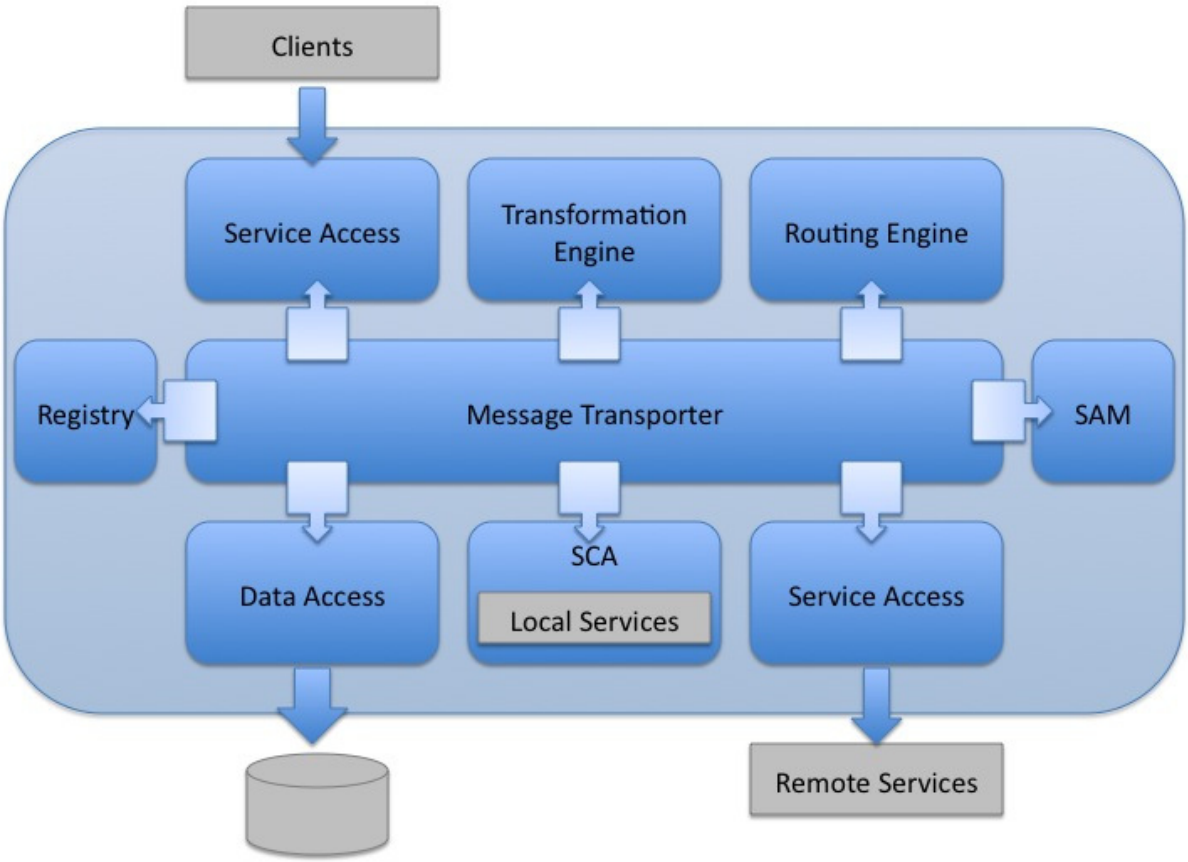

Figure 2 - SOA Architecture

- The message transporter insures the transmission and the message security between the various servers/hosts which are taking part in the consumers and providers deployments.
- The protocol mediators (Service Access) are used by the ESB to support various communication protocols between itself and the service consumers and providers.
- The routing engine insures the messages to be routed to the good recipient based on rules.
- The transformation engine insures the syntaxic message conversion between two protocols. It can also provide a semantic message conversion between the

consumer and the provider.

Calling a service

The following figure illustrates the difference between a simple service call and a service call through the ESB:

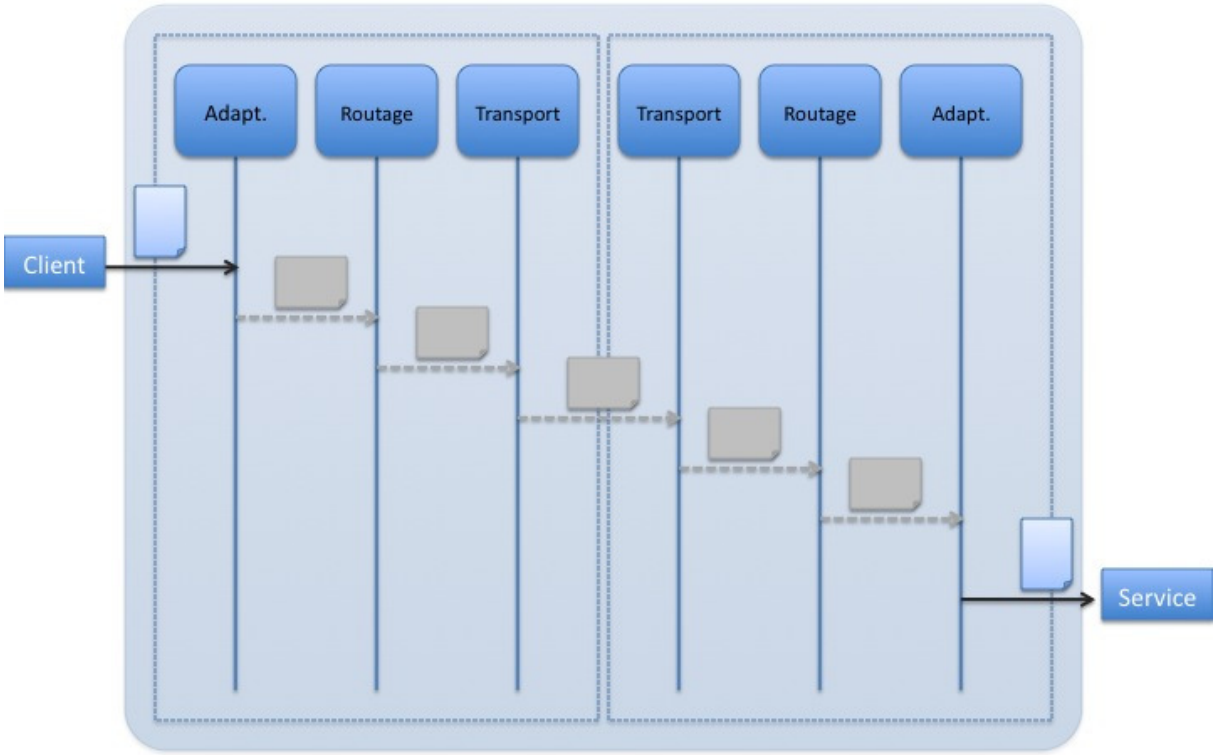

Figure 3 - Calling a Service

The client call is outed from what can be called a virtual service to the real service through the ESB following these steps (example between parenthesis):

1. A message is sent by the client to a virtual service. This service is linked to the ESB container by an adapter (ex: A JMS client posts a JMS message to a Topic, the new message is detected by a listener).
2. The adapter transforms the incoming message to an ESB internal message and sends this message to the routing engine (ex: The JMS message is transformed to the internal ESB format).
3. The routing engine selects the ESB service to call (ex: This can be based on the message content or by rules defined at configuration time).
4. The internal message transporter transports the message (with or without security, hopefully with security…) to the right application server node (ex: if the ESB is distributed over several nodes, the transport layer is in charge of sending the message to the wire).

5. Once received by the transporter, the message is going through the router and transmitted to the right adapter which is linked to the real service.

6. The message is transformed to the output format by the adapter (ex: The internal message is transformed to a SOAP message).

7. The adapter sends the message to the service (ex: the SOAP message is sent to a Web Service).

8. If the real service returns a response, this response is sent back to the service client through the ESB…

The main question is why do we need to add an ESB as mediator instead of calling the service directly? Here are some good reasons (this is not an exhaustive list):

- The client application does not have to wait the service response.

- The call can be securized by the ESB without any modification on the service side.

- The consumer delegates the service choice to the ESB. This really decouples the service consumers and providers.

- Decoupling service can hide the real service which will be called to the external service consumer. This contributes to securing access to the information system.

- The ESB offers features like monitoring (and more) which are not provided by simple solutions like standard Web Service calls.

- Service Level Agreement (SLA) can be defined between client and service. SLA is a part of a service contract where the level of service is formally defined (for example, limit the number of request per second or define the maximum response time which is accepted on the user point of view).

PEtALS Enterprise Service Bus

PEtALS (http://petals.ow2.org) is an Open Source (LGPL License) Enterprise Service Bus (ESB) provided by the OW2 middleware consortium (http://www.ow2.org).

PEtALS ESB is build with and on top of agile technologies such as:

– The Java Business Integration (JBI) v1.0 specification (http://www.jcp.org/en/jsr/detail?id=208). This is the Java standard for enterprise application integration. Note that in 2008, PEtALS has been certified by SUN Microsystems as a valid JBI implementation.

– The FRACTAL Software Component Framework provided by the OW2 consortium. Fractal is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces.
On the PEtALS point of view, all the container services (such as service registry, message router, message transporter, discovery etc...) are provided by the Fractal framework. This is a major feature which allows core developers to specialize a

PEtALS distribution by choosing the software components to be used for specific needs.

PEtALS is not only a JBI container, the project also contains tools for management, monitoring, frawework to create JBI components to hide the JBI complexity and a collection of Binding Components and Service Engines. All of this will be detailed in the next chapters.

Java Business Integration

The following section introduces the high level concepts of the specification which will be used next in the PEtALS related parts. For more advanced details, please refer to the specification.

The JBI specification has been standardized by the Java Community Process (JCP) expert group in the JSR208 document.

The specification defines a standard means for assembling integration components to create integration solutions that enable a SOA in an enterprise information system.

Environment

Components are plugged into a JBI environment and can provide or consume services through it in a loosely coupled way. The JBI environment then routes the exchanges between those components and offers a set of technical services. JBI is built on top of state-of-the-art SOA standards: service definitions are described in WSDL format and components exchange XML messages in a document-oriented-model way.

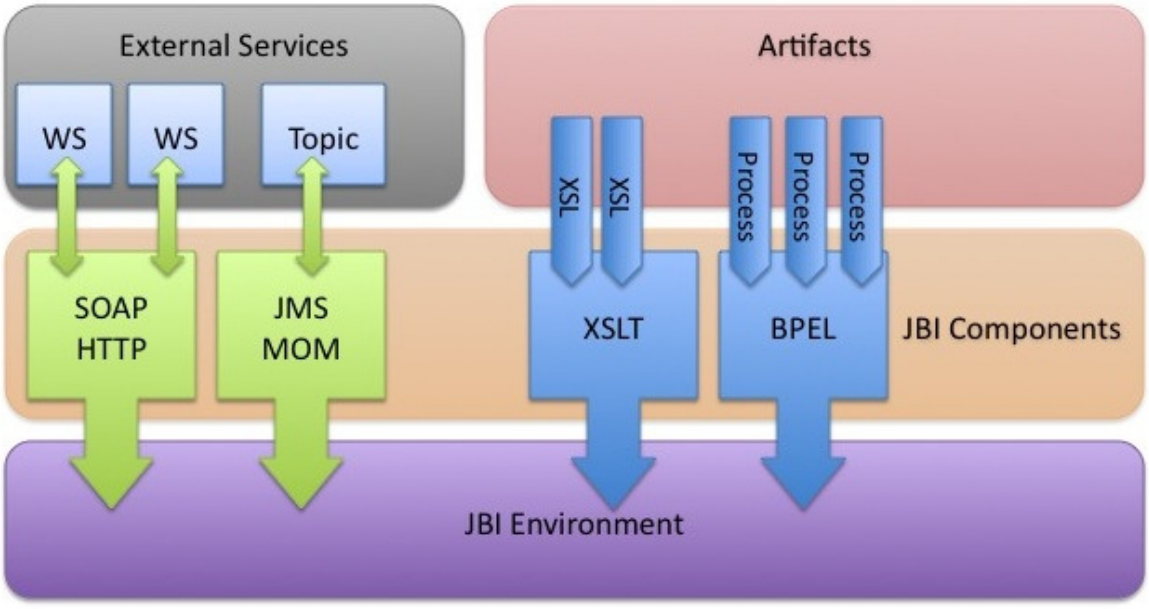

Figure 4 - JBI Architecture

The central part of the JBI specification is the Normalized Message Router (NMR), described as the "JBI Environment" in figure above. The NMR ensures loosely coupled communication by providing standard Service Provider Interfaces (SPI) that promote the exchange of XML documents between plugged JBI components and loose references

via the use of their interface name.

JBI Artifacts

The JBI specification defines a set of artifacts which are used to add connectivity, bind/expose services and configure the Service Bus.

The main artefacts are the JBI components which are divided in two families:

- **Binding Components** (BC) – "connectors" which are used to interface the JBI bus with the rest of the Information System (Green boxes in figure above). Binding Components enable both the exposure of external resources in the bus and the exposure of services available on the bus for their use by external consumers: e.g. connections to Web services, FTP, Mail, Message-Oriented Middleware, or even standard business communications like EDI or ebXML.

- **Service Engines** (SE) provide the integration logic (Blue boxes in the figure above). They typically handle messages that pass through the bus in order to provide routing (e.g. content-based routing, priority-based routing), transformation (XSLT), orchestration (BPEL), log or audit features.

In order to activate endpoints in the JBI environment, artifacts named **Service Units** (SU) must be deployed on the JBI component (both BC and SE). The SU contains configuration file which are used by the JBI component consume or provide a JBI service. A SU has two modes:

- Consumer: The component on which the SU is deployed on will consume the service described in the SU configuration file.

- Provider: The component on which the SU is deployed on will provide the service described in the SU configuration file.

Service Units are packaged in artifacts named **Service Assembly** (SA). The SA can contain a collection of SU plus a configuration file which describe on which component each SU must be deployed.

The last artefact defined in the JBI specification is the **Shared Library** (SL). The SL is an artefact which can be shared between JBI components. The components will potentially use the Java libraries and resources bundled in the SL to create their class loaders.

In this document, a JBI artefact will refer to the previously detailed items. More details on Binding Components, Service Engines, Service Units, Service Assemblies, Shared Libraries, their lifecycles and usage are described in the JBI specification.

Messages

The atomic JBI message, also called Normalized Message, is composed of a XML payload, attachments (binary data) and key-value properties (also view as message context). A JBI message transits between service consumer and provider in a Message Exchange. A Message Exchange contains an input Normalized Message, a set of key-value properties and potentially:

- An output Normalized Message representing the service invocation response. This depends on the Message Exchange Pattern (MEP) used to invoke the service.

- A Fault (extension of a Normalized Message) when something wrong occurs on the service side.
- A Pattern
- A Status
- …

The Message Exchanges can be classified by patterns (inspired by the WSDL 2.0 specification); we talk about Message Exchange Pattern (MEP):

- InOnly: This pattern is used for one-way exchanges
    - The service consumer sends a message
    - The service provider replies by an aknowledge
- InOut: This pattern is used for two-way exchanges.
    - The service consumer sends a message
    - The service provider reploies by a message or a fault
    - The service consumer acknowleges
- InOptionalOut: This pattern is used for a two-way exchange where the provider's response is optional
    - The service consumer sends a message
    - The service provider can send back a response, a fault or just an acknowledgement. The consumer:
        - Acknowledges the response
        - Or sends a fault, in this case the provider must acknowledge this fault
        - Or acknowledges the initial fault
- RobustInOnly: This pattern is used for reliable, one-way message exchanges
    - The service consumer sends a message
    - The service provider acknowledges or send back a fault
    - If a fault is returned, the service consumer acknowledges

More details on section 5.4.2 of the JBI specification.

Delivery channel

The Delivery Channel (DC) is the interface between the components and the Normalized Message Router:

- The consumer (a JBI component) sets the message exchange (payload,

attachments and properties) and send the message to the delivery channel

- The message bus is in charge of transmiting the message to the component which is providing the requested service
- The provider (a JBI component) receives the message exchange from its delivery channel and processes the message. Depending on the message pattern, a response, fault or acknowledgement can be set in the message exchange. The message exchange is then sent back to the consumer through the delivery channel.

PEtALS High Level Architecture

The PEtALS architecture has been defined using the software component paradigm. The following figure introduces the main components which are required to build the PEtALS container.

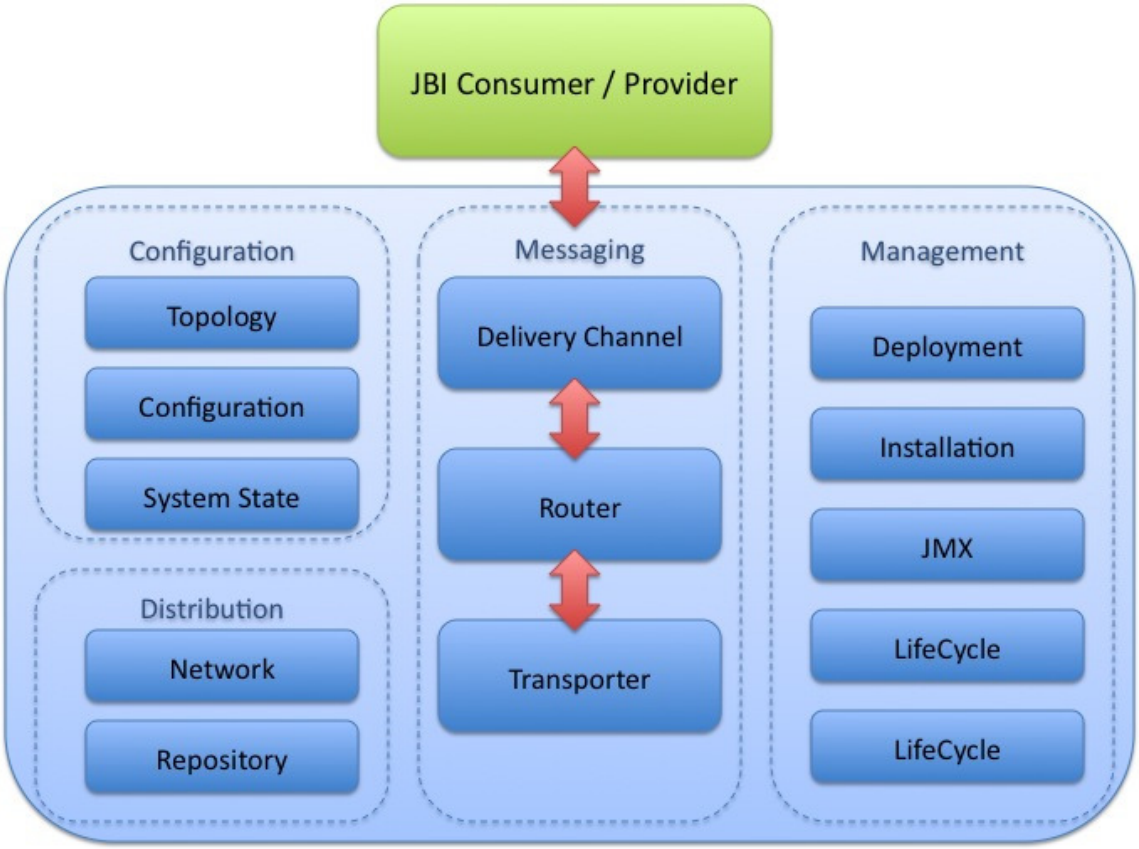

Figure 5 - PEtALS High-Level Architecture

Details about the software components are detailed in the next section.

Software components

The container kernel is build upon the software component paradigm. The Component Based Development (CBD) emphasises on decomposition of the engineered systems into functional or logical components with well-defined interfaces used for communication across the components. Components are considered to be a higher level of abstraction than objects and as such they do not share state and communicate by exchanging

messages carrying data.

By assembling components implementations, it is possible to specialize a software for different needs. In the PEtALS container context, we can easily replace a complex distributed registry by a local hastable and provide a simple standalone PEtALS distribution (plus some distributed components). This is one of the interesting features the software component model provides.

One other powerful feature will be the possibility to replace software components at runtime. Even if this approach will never be used in production environment, let's imagine that you need to replace the JMS based communication channel by a HTTP one because your network topology or firewall settings have changed. This will be possible without stopping the container. Just stop the component (all the calls to this component are buffered during the operation), undeploy it then deploy and start the new component which provide the same interface. There is no impact on the container and all the buffered messages are then send to the new component!

On the PEtALS developer point of view, working with Fractal components is like working with Java interfaces, annotations and descritpor files.

```
@FractalComponent
@Provides(interfaces = @Interface(name = "service",
  signature = org.ow2.petals.jbi.management.installation.InstallationServiceMBean.class))

  public class InstallationServiceImpl implements InstallationServiceMBean {

    @Requires(name = "configuration",
      signature = org.ow2.petals.kernel.configuration.ConfigurationService.class)
    private ConfigurationService configurationService;

…

}
```

This code snippets taken from the PEtALS kernel sources specifies that :

- The InstallationServiceImpl is a Fractal component (FractalComponent annotation).
- The Fractal component provides a service to other components which is defined by the InstallationServiceMbean interface (Provides annotation) and which is implemented in the current class.
- The InstallationServiceImpl need to use a service which provides the ConfigurationService interface.

Since the Fractal component implementation provides and requires services implementation, all the component implementations are binded together using XML descriptor files. This descriptor file is processed at compile time is used to generate the final Java byte code.

The OW2 Fractal component model has been chosen for its very structuring architecture and is used in the PEtALS container to separate and identify modules. Fractal is not limited to the features introduced above. The framework is really powerful; we really encourage users and developers to have a look to the documentation which can be found on the Fractal Web Site (http://fractal.ow2.org).

PEtALS Components

As described in the previous section, PEtALS is built using the software component approach. The main PEtALS components are listed below:

- The **Container Service** (*org.ow2.petals.container.ContainerService interface*) is the main container component. Its role is to create the JBI artefacts lifecycles, installers. In the kernel point of view, the resources created by this service are wrapped into Fractal components.

- The **Router** (*org.ow2.petals.jbi.messaging.routing.Router interface*) acts as the central part of the JBI container. The router is in charge of choosing the right endpoint to send the JBI message to.

- The **Transporter** (*org.ow2.petals.transport.Transporter interface*) is the mediator between the router and the communication channel. Its roles are:

    - Sending messages to the endpoint resolved by the NMR. As explained in the inter node communication section, sending a message to an endpoint can be just an in memory process or if the endpoint is located on a foreign container, the message is serialized into the wire format then send to the foreign container.

    - Receiving messages which is exactly the opposite of the sending role.

- The **Endpoint Registry** (*org.ow2.petals.jbi.messaging.registry.EndpointRegistry interface*) is in charge of storing the endpoint references and their service descriptions (WSDL descriptions). The endpoints are classified in a tree form (interface, service and endpoint branches) in order to optimize their retrieval.

- The **Network Service** (*org.ow2.petals.communication.network.NetworkService interface*) is responsible of managing nodes whi want to join or leave the PEtALS network. *NOTE: This service must handle the state of the remote nodes by sending hello/ping messages…*

- The **Topology Service** (*org.ow2.petals.communication.topology.TopologyService interface*) holds all the PEtALS nodes configuration. The complete topology information of the PEtALS network is updated when nodes are joining or leaving the network.

- The **Configuration Service** (*org.ow2.petals.kernel.configuration.ConfigurationService interface*) handles the configuration of the local PEtALS container. This local configuration (class org.ow2.petals.kernel.configuration.ContainerConfiguration) contains data for the container such as topology, server properties (name, timeouts, QoS, JAAS, SSL…) but also domain and subdomain informations.

- The **Repository Service** (*org.ow2.petals.platform.repository.RepositoryService interface*) handles JBI artefacts resources needed at runtime by the container. When a JBI artefact is installed/deployed on the container, it is exploded in a distinct folder holded by the Repository Service.When the container is restarted, all the JBI resources are reloaded from their repository path. This allows JBI artefacts such as components to store/persist resources to be reused in their

dedicated workspace.

- The **System State Service** (*org.ow2.petals.platform.systemstate.SystemStateService interface*) is in charge of storing states (lifecycle state, installation URL, source URL, …) and maintening the consistency whenever a JBI artefact state is updated. The system state and the repository services work together to reload resources on container reload.

- The **JMX Service** (*org.ow2.petals.communication.jmx.JMXService interface*) is used to manage JMX connections to a local or a remote PEtALS container. The JBI specification defines a set of JMX operations to be exposed in order to manage a JBI container such as installation, deployment, lifecycle management, … (details on Chapter 6 - JBI specification).
  This service is the local access point to all the remote containers JMX operations. By using this service, the local container can call these operations and so manage all the remote containers.

- The **JNDI Service** (*org.ow2.petals.communication.jndi.client.JNDIService interface*) provides facilities to connect to a JNDI repository (http://java.sun.com/products/jndi/).
  In PEtALS 1.x and 2.x, the JNDI repository is used to share knowledge between nodes such as JBI Endpoints and container configuration. This is a central point of the PEtALS environment and you are, of course, free to choose the JNDI repository provided by PEtALS or a third party one. This is possible by setting the JNDI factory properties at the domain level in the topology configuration.
  PEtALS 3.0 comes with a new approach where the JNDI repository is replaced by an extensible and configurable distributed repository.

- The **Installation Service** (*org.ow2.petals.jbi.management.installation.InstallationServiceMBean interface*) role is to manage the installation (and provide the the lifecycle information) of the JBI components and shared libraries.

- The **Deployment Service** (*org.ow2.petals.jbi.management.deployment.DeploymentServiceMBean interface*) role is to manage the deployment (and provide the lifecycle information) of the service assemblies. The service units bundled into the service assemblies are deployed to the right component by this service implementation. As a result, JBI service consumers and providers are activated in the JBI and PEtALS environment.

The links between these components are at the charge of the PEtALS kernel developer. The final assembly of different component implementations will result to various PEtALS behaviours. This is what it is used to created specific PEtALS distributions (see PEtALS distributions section).

The Fractal architecture of the PEtALS container can be found on the Annex A.

PEtALS JBI Implementation

The current section introduces how JBI artifacts are used in the PEtALS JBI container and how they help to instanciate the SOA.

The PEtALS container is based on the JBI 1.0 specification and has been certified as a JBI compliant container by SUN MicroSystems. This certification has been obtained by running the official SUN JBI test suite with PEtALS as JBI runtime. This suite tests that all the features defined in the specification are well implemented.

In this section, a 'container service' is a software service which is provided by the container like, for example, the message routing, the exchange security. A 'JBI service' is a service created from a JBI artefact which is used to send, receive and potentially process JBI messages.

The PEtALS JBI Container

The JBI container is a Java based library which can be used in several forms such as standalone server, embedded in an entreprise application server, embedded in a standard Java application, … The JBI container is generally called 'kernel' in PEtALS and implements all the JBI specification.

There is no container lifecycle defined in the JBI specification. Starting PEtALS instantiates all the software services needed to welcome the JBI artefacts, exchange messages between JBI services and some additional features which will be detailed later in this document.

The JBI container can be seen as a service container and like a web application container, a JBI container is not useful when used alone. To use the container, artifacts needs to be installed/deployed into the container with the help of management operations.

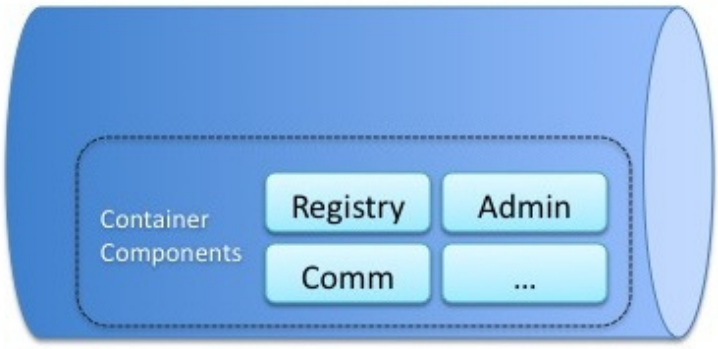

Figure 6 - PEtALS Container

Using a JBI Component

The PEtALS components are packaged as ZIP archives which contains the following files:

```
META-INF/jbi.xml
petals-COMPONENT-NAME.jar
a.jar
b.jar
```

COMPONENT-NAME is the name of the component. This JAR file contains the classes which implements the JBI component interface. The other JAR files are third party libraries which are used by the component implementation.

The JBI component must be installed in the JBI container. The container handles the component and its descriptor file (the jbi.xml file) which defines JBI properties such the name, the type (binding component or service engine) and component specific properties. The component properties will ne used by the component implementation to configure the component (a pool size, a socket address, ...).

The Java classes, which are packaged into the component, are used to create the component class loader and the container links the component to the container with a context and the delivery channel used to send and receive JBI messages. Once these steps are achieved, the component is started but it can not receive and process JBI messages.

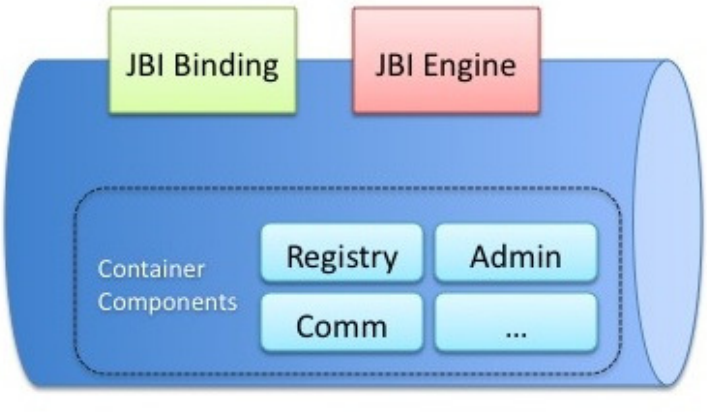

Figure 7 - PEtALS Container and JBI Components

Using a JBI Service Unit

The PEtALS Service Units are packaged as ZIP archives which contains the following files:

```
META-INF/jbi.xml
Service.wsdl
```

The service unit is used to activate consumer or provider endpoints on JBI components. When the service unit is deployed on the container, this one will create an endpoint with the information provided in the service unit descriptor (jbi.xml file). A service provider endpoint is defined by a service name, an interface name, an endpoint name, a WSDL description (locally from the service unit or remotely with an URL) and some additional properties.

Once deployed, the messages, which are sent to the endpoint, will be delivered to the component the service unit is deployed on. It is up to the component to process the message and to return a message, a response, a fault, or nothing.

Processing a message is specific to the component implementation and is not defined in the service unit:

- A binding component will generally transform the message into another format (for example to a SOAP message) and do its job (for example invoking an external Web Service).

- A service engine will get the message and processes it (orchestration, transformation, apply rules). The engine will also potentially invoke other JBI services.

The service unit can not be used alone since the component to deploy the service unit onto is not defined in its descriptor.

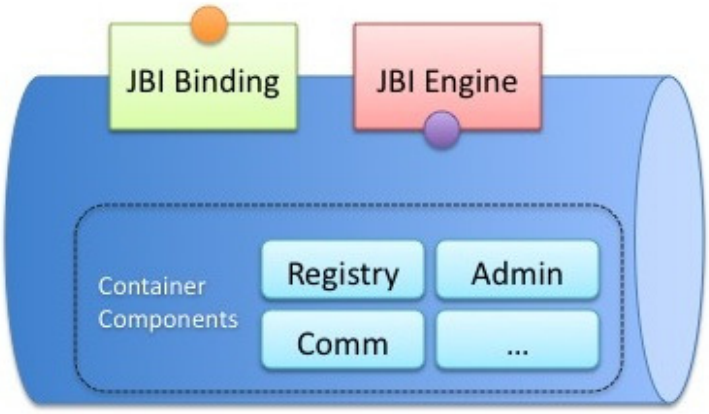

Figure 8 - PEtALS Container and JBI Endpoints

Using a JBI Service Assembly

The PEtALS Service Assemblies are packaged as ZIP archives which contains the following files:

```
META-INF/jbi.xml
su-A.zip
su-N.zip
```

The service assembly contains a collection of service units and a JBI descriptor file (jbi.xml under META-INF folder). Each service unit is deployed onto the component which is defined in the service assembly descriptor file. By deploying the service assembly into the container, it will get all the service units and will try to deploy each one on the sepcifed component.

Using a JBI Shared Library

The PEtALS shared libraries are packaged as ZIP archives which contains the following files:

```
META-INF/jbi.xml
lib-A.zip
lib-N.zip
```

The shared library contains a collection of Java libraries and a JBI descriptor file (jbi.xml under the META-INF folder). The descriptor file gives the description of the shared library (its name, version) and the list of embedded Java libraries.

A shared library is deployed into the container. Once deployed, this artifact may be potentially used by components to enrich their class loader. This type of artefact is generally used by components which uses the same libraries and so to limitate the resources in the PEtALS container.

Using the JBI environment

Now that the container is started, the components are installed and the endpoints are activated with the help of the service unit deployment, services can be invoked and messages can be exchanged between service consumers and providers.

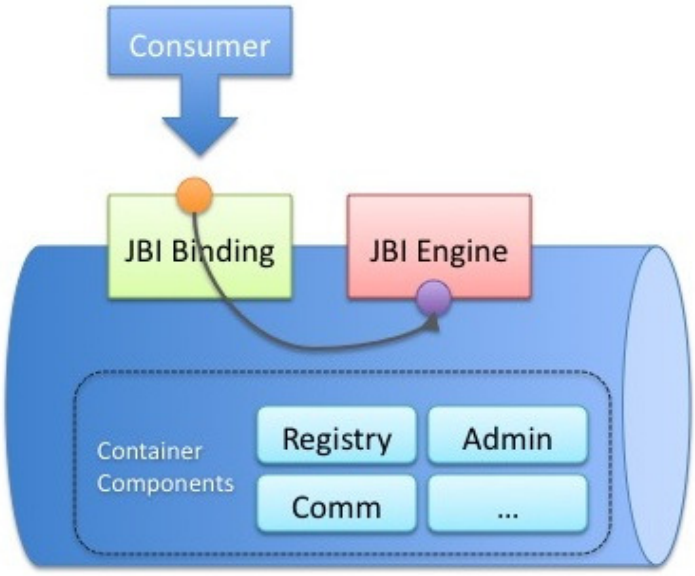

Figure 9 - Using PEtALS

In the previous figure, a web service client calls a web service which is hosted by the JBI container. This web service is just a facade. The JBI service which is consumed by the facade can be changed as many time as needed. The only condition is to provide the same interface. This approach is extremely flexible since the JBI service can be a service composition of other JBI services (and recursively to reach atomic JBI services).

PEtALS JBI Extensions

PEtALS is not a simple JBI implementation, the current chapter introduces specificities and additionnal features which are not defined in the standard specification.

Distributed Environment

The JBI specification does not deals with any distributed aspect. The main JBI extension and PEtALS main feature is the distributed approach.

A PEtALS container can share its services and access services which are hosted by other containers in a transparent way. It means that on the service consumers and providers' point of view there is no additional configuration like in other JBI containers. Where other JBI containers provide a distributed approach by connecting containers with the use of JBI Binding Components plus huge configuration, PEtALS provide this feature natively without any additional configuration.

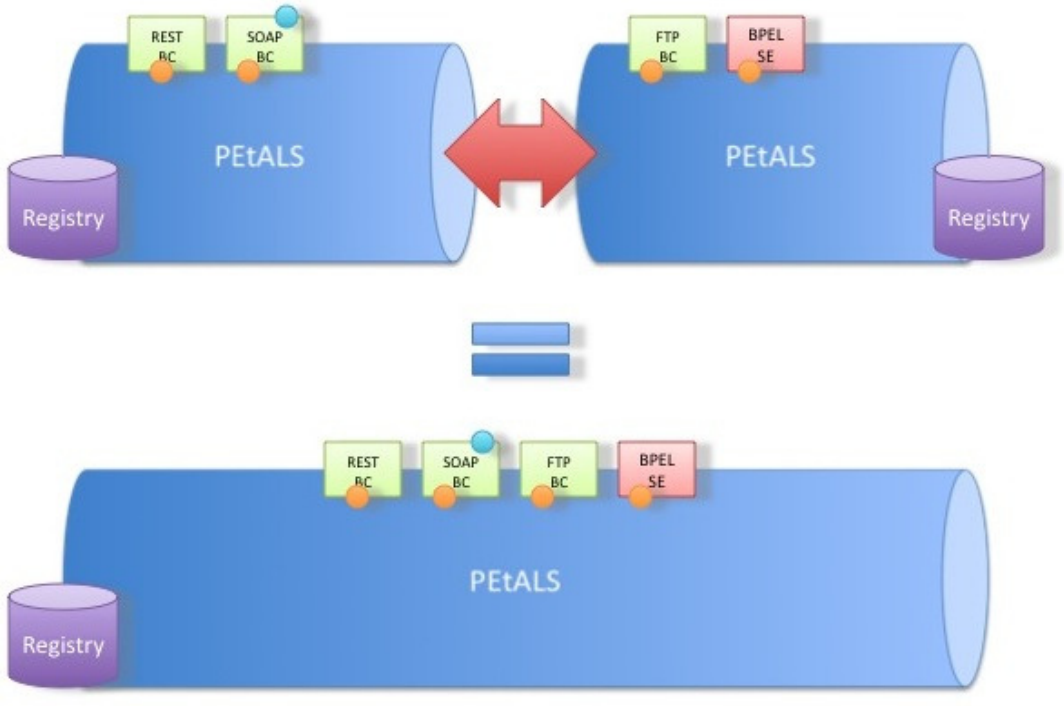

Figure 10 - Distributed Service Bus

The previous figure shows that multiple containers with their deployed JBI artifacts are completelu equal to a single container which holds all these artefacts. The following sections introduce the software modules which are needed to build this Distributed Entreprise Service Bus.

The advantages of this distributed environment are:

- Container and connectors can be deployed close to the real services. This is a way to bypass communication constraints between JBI consumer and JBI provider like firewall issues and other network constraints.
- Complete control of the inter-container communication layer. As described in the previous point, each link between consumer and provider may have a specific configuration. Opening communication ports for JMS, HTTP, FTP, and more is never possible. Opening and securing only communication for one protocol is enough for the PEtALS inter-communication protocol. A SOAP service client will be able to post a JMS message into a topic without any http or JMS port opening.

Technical Registry

The PEtALS JBI services, endpoints, interfaces, WSDL descriptions and container location are stored in an embedded technical registry. This registry is used by the PEtALS containers to register services and to route the JBI messages to the right endpoint.

In order to have a unified vision, the registry entries are replicated among all the PEtALS nodes using a Distributed HashTable over a multicast channel. This is quite equivalent to data flooding between registries so when an entry is added to the registry, the data is send to all the network registries. All the registries have a complete vision of the services hosted by all the containers.

Note: Since the replicated approach is not scalable at all, the PEtALS v3.0 will come with a new distributed and extensible registry.

Inter nodes communication

In order to exchange (send/receive) messages between PEtALS containers, PEtALS extends the JBI specification by introducing a message transporter layer.

In a standard JBI implementation, the Normalized Message Router gets the local endpoint reference from the local registry and sends the message to a local JBI endpoint. In the PEtALS approach, once the endpoint is retrieved from the local registry, the message and the endpoint reference are sent to the transport layer which is in charge to deliver the message to the JBI endpoint whereever the container it is hosted on. This is possible by getting the foreign container information where the JBI services is hosted on from the technical registry. Once all the required information is retrieved, PEtALS serializes the JBI message to the 'wire format' and send it to the foreign container. The response is sent back from the remote container is needed.

All the containers are linked together by the transporter layer. All the services providers and consumer scan potentially send messages to each other. All the links which were at the charge of the distributed application developer are now hidden by the PEtALS container not only at the service level but also at the communication one.

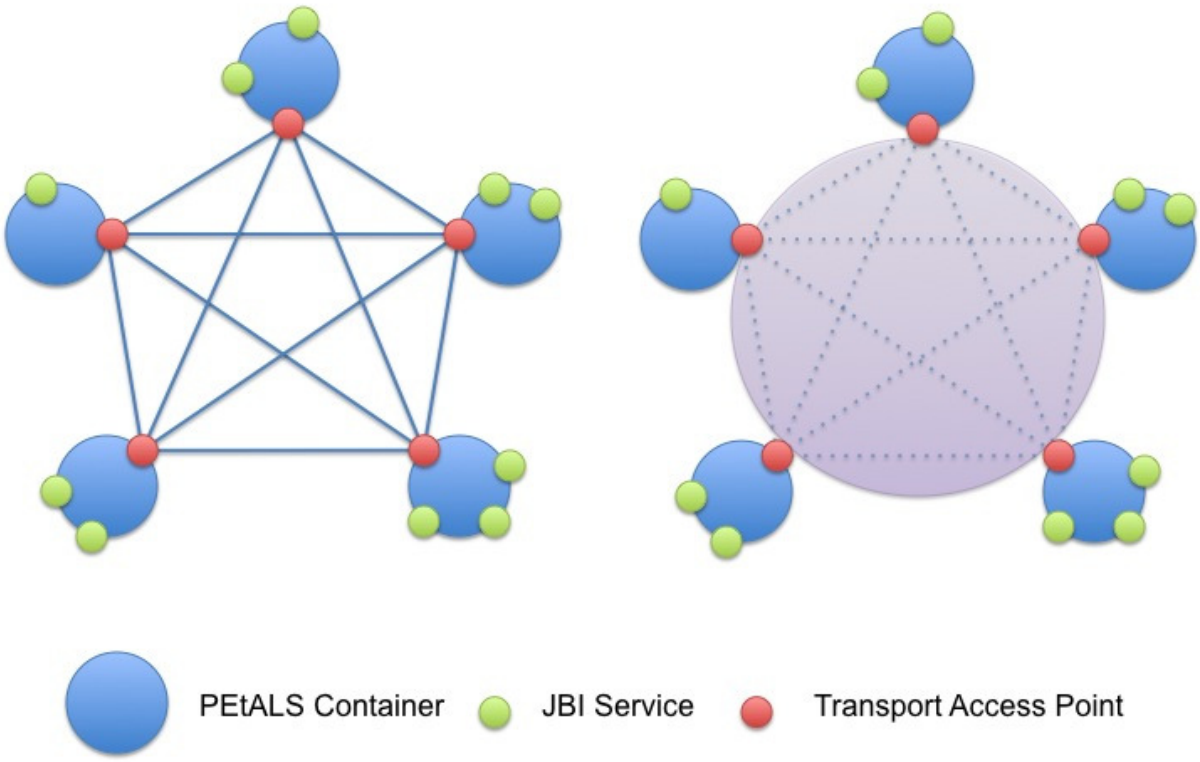

Figure 11 - Inter nodes communication

The previous figure illustrates two visions of the inter node communication which are totally equivalent. On the left side, all the containers are linked together by point to

point communication channels. On the right side, all the containers can also communicate to each other but the links are totally hidden by an abstract communication layer. At the lower level, the links exist but are not important on the transport user point of view (in the PEtALS case, the router layer does not care about links between containers).

JMS is the historical messaging system PEtALS uses to exchange messages between hosts (OW2 JORAM JMS implementation http://joram.ow2.org). Since PEtALS 2.0, a new message transporter has been added. The OW2 DREAM project (http://dream.ow2.org) used for this transporter implemantation is a component-based framework dedicated to the construction of communication middleware. It provides a component library and a set of tools to build, configure and deploy middleware implementing various communication paradigms: group communications, message passing, event-reaction, publish-subscribe, etc... DREAM is alo build upon the FRACTAL component framework, which provides support for hierarchical and dynamic composition.

This communication abstraction is also important on the JBI service consumers and providers side. When standard applications need to create links between service consumers and providers (for example an application which wants to get weather forecast from a Yahoo service and get stock values from the Google service) needs to basically create the links between the client (application under development) and services (Yahoo, Google, and more). Of course, the ideal solution wil be to add an abstraction layer to all of this in case of service provider modification (URL, interface, …). A solution such as an ESB already offers this abstraction layer since the links are the container responsability.

Management

A specification section (chapter 6) is dedicated to the container management using JMX. It mainly deals with the JBI artefacts management such as installation, deployment, starting and stopping. Additionally, the specification defines a set of Apache Ant (http://ant.apache.org) taks to manage the container from Ant scripts.

PEtALS implements and extends the management requirement of the specification in the way that additional JMX operations have been added to the specification ones. This is possible by extending the Managed Beans (MBeans) and adding new ones.

Note : Must group all the MBeans in a specific package (org.ow2.petals.core.kernel.management) and list them.

Monitoring

PEtALS provides various types of monitoring features:

- Use JMX to store and expose monitoring data. This approach uses the JMX features provided by the container and adds specific monitoring MBeans.
- Use the Fractal Framework. Since quite every JBI artefact (JBI component, JBI endpoint, …) and container module are Fractal components, PEtALS uses the Fractal potential to provide very advanced monitoring data.

There is actually some work to generalize these monitoring approaches to all the container software components and also to provide monitoring data in more standard

ways (using OASIS WSDM specification and WS-* seems to be good candidates). The following figure is a general vision of what is planned by adding aspects/wrappers/controllers to Fractal software components:

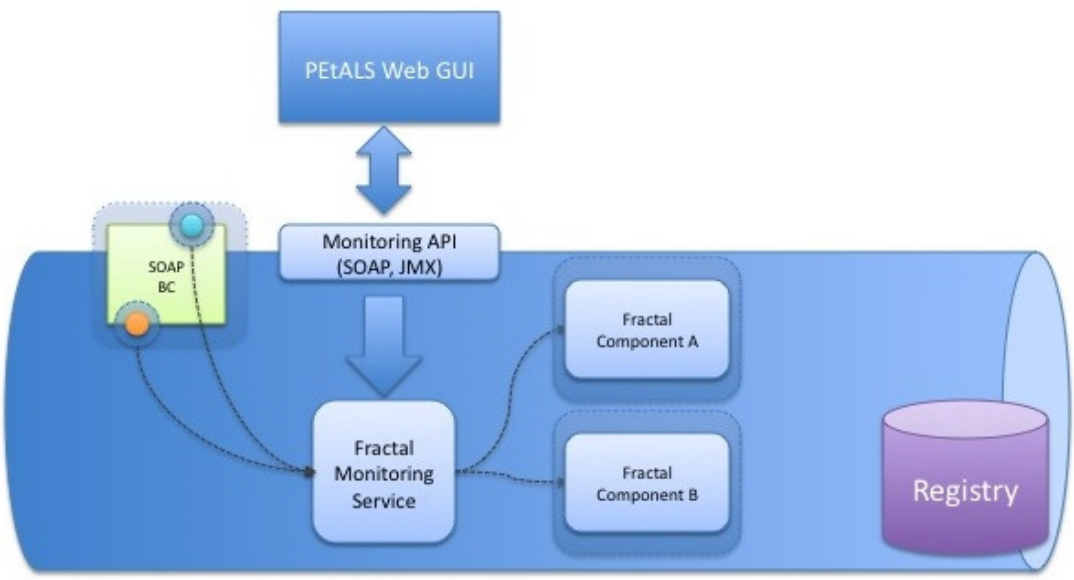

Figure 12 - Global Monitoring Vision

The main usage if this monitoring data is to display JBI messages and classify them by service, endpoint, interface, container, component, … A tool such as the PEtALS Monitoring and Management WebConsole uses this monitoring data and provide it in a nice web enabled GUI (of course users can develop their own monitoring client).

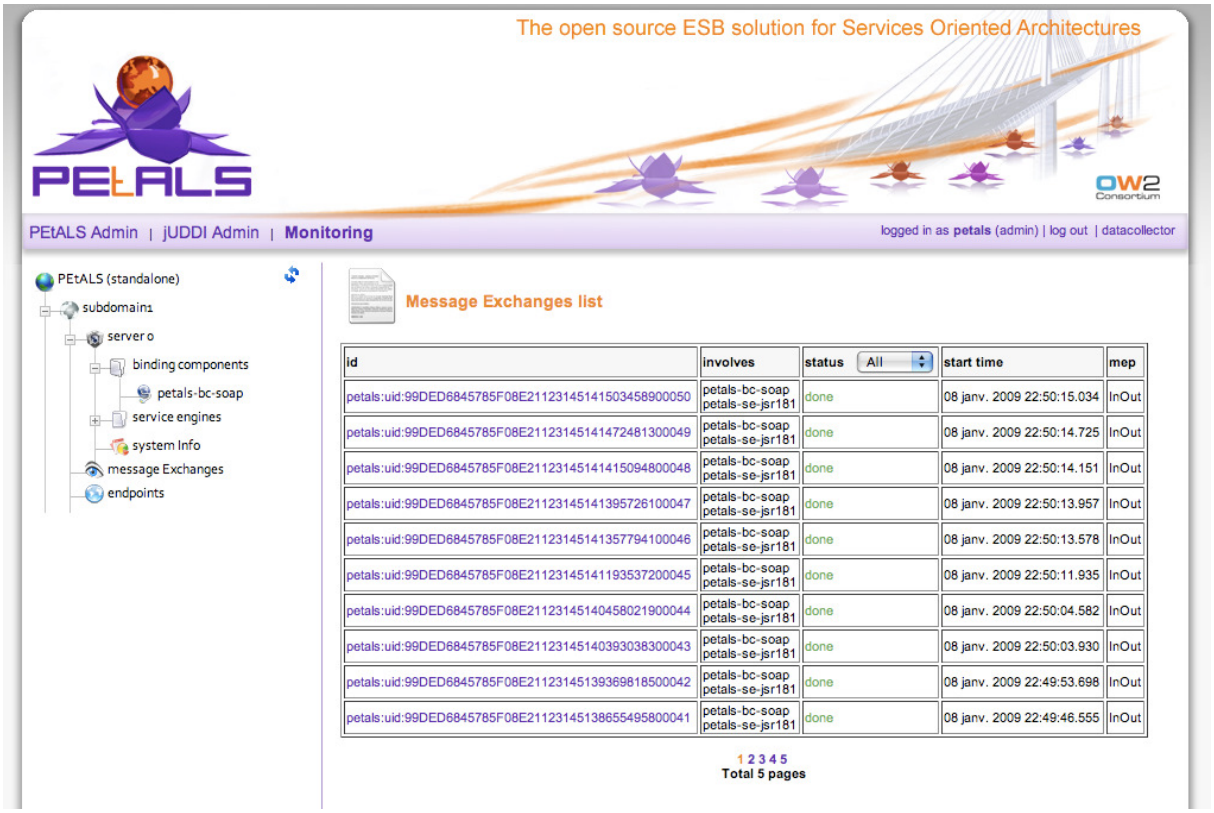

Figure 13 - Message Monitoring

Another usage of the monitoring data is for load-balancing. Since PEtALS is a distributed container, the monitoring data can be a good candidate to choose the right JBI endpoint depending on its load. More details are given in the load-balancing section.

Kernel Features

The kernel fully implements the JBI specification. Some features have been introduced in the previous chapters, other ones are defined below.

Domains

PEtALS is not a simple distributed JBI container where all services can be accessed by all the consumers. To add some visiblity control containers can be classified by domains and subdomains.

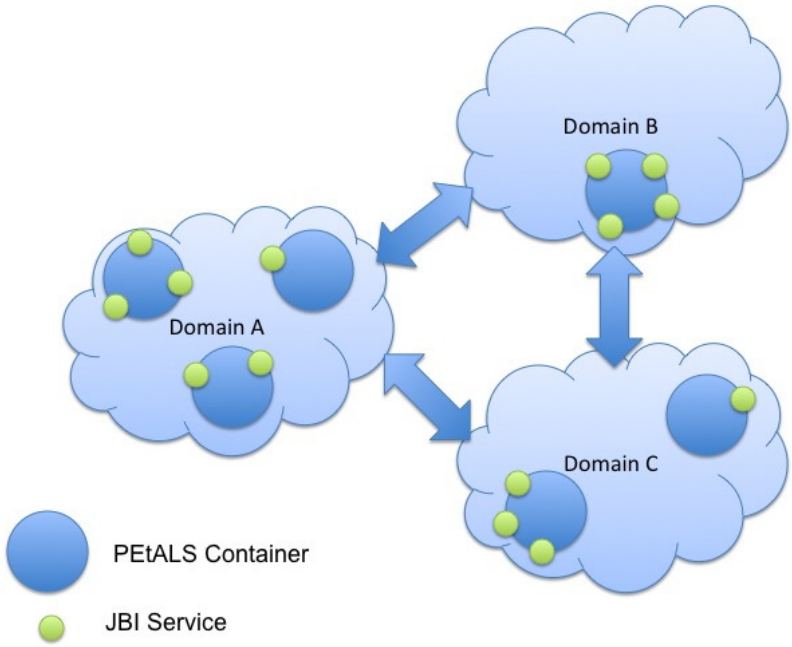

Figure 14 - PEtALS Domains

- PEtALS containers can only communicate with containers which belong to the same domain.
- Services can be declared as public or private so that private services are only accessible from containers which belong to the same subdomain.

This domain and subdomain settings are basically called 'topology' and are defined in the topology.xml file under the PEtALS distribution configuration folder.

JBI exchange optimisation

JBI message responses, faults or acknowledgements generates extra trafic between consumer and provider because the JBI container must transfer all the message exchange content (the input message, the output one, …) at each pattern step. It is clear that serializing this data and sending it to the wire are time consuming in the distributed environment.

By setting the *org.ow2.petals.messaging.noack* property on the message exchange, the acknowledgement and error messages will not be sent through the NMR.

Address Resolver

The Adress Resolver acts as the router level (embedded into a router module) to select the right endpoint to send the message to.

To select the right endpoint, the AR interacts with the Endpoints Registry to get all the endpoints which provide a valid interface and filter the endpoint list by applying a strategy (*org.ow2.petals.jbi.messaging.routing.strategy.Strategy interface*).

The strategy is defined at the Message Exchange level (*org.ow2.petals.routing.strategy* property) set by the service consumer. Current strategies are :

- **random**: Randomly select an endpoint from the input endpoints list.
- **default**: Select a default strategy from the available ones. No comments in the code…
- **standalone**: Randomly select an endpoint from the input endpoints list.
- **highest**: Select an endpoint based on some weighting given in the message exchange property.

Note: The strategies are defined at the code level and there is no way to add user ones. This code has to be refactored to define strategy in a configuration file in order to give the user the possibility to ad dits own strategies. JAVADOC needs to be added for the current strategies.

Router Module

The router module has been decomposed into atomic modules (*org.ow2.petals.jbi.messaging.routing.module.Module interface*).

Once the router receives a JBI message from the Delivery Channel or from the Transport layer, it injects this message sequentially in each module for processing.

The standard modules are:

- The **address resolver module** (AR): The AR selects the good endpoint to send the message to.
- The **transport module:** In charge of communication with the transport layer to send/receive messages.
- The **authorization module**: Checks that the message can be send to the selected endpoint. This module uses JAAS for access authorization.

These modules are activated (or not) in the *router.cfg* configuration file located under the *conf* folder of the PEtALS distribution. This part has been designed to be extended by your own module to add some functionnalities at the router level (logging, reporting, …).

Transport layer

The Transport layer, which is in charge of sending/receiving messages to/from the wire, is electing a message transport from the message exchange properties.

Selecting the right message transporter depends on the QoS the service consumer has defined in the org.ow2.petals.transport.qos message exchange property. Possible values are:

- **fast**: Assures that the message will be sent in the fast way. The message can be lost by the local container if something wrong happens and i twill not be possible to send it again.
- **reliable**: Assures that the message will be persisted in order to be ressent if something wrong happens.

In PEtALS v2.x, the technologies used to transport messages are OW2-JORAM (open

source JMS), OW2-Dream (open source Extensible Messaging Framework, used in TCP/IP mode) and 'InVM' (Memory transport in the same Java Virtual Machine).

Security

Transport level security

Since messages are exchanged between nodes, there is a need of securing the data transmission. This is actually possible by using a SSL channel in the OW2-Dream based message transporter.

Authorization

It is possible to define authorizations at the service operation level.

A router module is in charge of the authorization; it checks that the user (which belongs to a group of users, and which is sending a message to the JBI service) is authorized. This is possible by using the JAAS technology (http://java.sun.com/javase/technologies/security/) in adition to the JBI message exchange security subject.

The default implementation of the JAAS authorization login module is based on a simple configuration file. This router module is configurable and developers can define their own authorization login module based on other users repositories (LDAP for example).

Hot Deployment

The kernel provides a hot deployment features to avoid any Ant script or JMX commands. This hot deployment feature allows to install and start JBI artifacts archives (Component, Service Unit, Service Assembly and Shared Library as ZIP files) by copying them in the distribution install folder. When the kernel detects a new artifact, it install/deploy it and process the lifecycle steps to effectively start the artifact.

The same feature is also available for the uninstallation/undeployment steps. By deleting the artifacts from the uninstall folder, the kernel detects that the artifact must be stopped and undeployed/uninstalled.

Behind the scenes, the kernel calls the JMX management operations needed to process all these steps.

These hot deployment/undeployment features are similar to the web application deployment feature you can find when copying a WAR file into the webapps folder of an Apache Tomcat container.

Data compression

Since JBI messages are XML based and potentially contain redundant data, the compression feature has been added when exchanging messages between PEtALS containers. This compression is done at transport level when a property (*org.ow2.petals.transport.compress*) is detected in the message exchange (set by the service consumer).

Source code

Modules

This section describes the main modules of the PEtALS project. Instructions about how to get the sources from the OW2 forge and build the project can be found on the PEtALS Developer Guide (http://forge.objectweb.org/docman/?group_id=213). These modules are divided in families, their descriptions are given below.

- **petals-cdk**: The PEtALS Component Development Kit is a framework used to easily create advanced JBI component wihout any JBI knowledge.
    - **petals-cdk-api**: The CDK API. Used by the CDK core module and the interceptors.
    - **petals-cdk-core**: The CDK core is the main CDK module which provide all the features to easily create advanced and perfromant JBI compatible component without any JBI knowledge. Refer to the CDK chapter to get more details.
    - **petals-cdk-interceptors**: This module contains some component interceptor implementations. Refer to the CDK chapter to get more details.
- **petals-commons**: Contains utilities and common classes used by other modules.
- **petals-components**: Contains all the PEtALS JBI components (Service Engines and Binding Components) provided by the project. Refer to the components section for more details on the available components.
- **petals-core**: This parent module contains all the modules used to create the container.
    - **petals-ant**: The ANT task implementation based on the JBI specification used for container management.
    - **petals-kernel**: The container implementation based on the JBI specification.
    - **petals-kernel-api**: The kernel API is shared by the core modules and the container launcher.
    - **petals-kernel-ext**: This module contains some kernel extensions which are not needed by the container to run. These extensions are generally added to the container assembly to build specific distributions. A good example of extension is the OW2-Dragon (http://dragon.ow2.org) connection implementation, which is added to the Dragon-enabled PEtALS distribution.
    - **petals-monitoring**: The monitoring module add some monitoring features based on the Web Service Distributed Management (WSDM) OASIS specification (http://www.oasis-open.org/committees/wsdm/).
- **petals-topology**: Used to manipulate the topology classes generated from XML Schemas.

- **petals-demos**: This parent module contains usecases and demos based on the PEtALS container, PEtALS components and service implementation such as Web Services, JMS Topics, etc…
- **petals-distribution**: This parent module contains modules used to build specific distributions (standalone, Platform, quickstart) based on various Fractal descriptor files. It also contains the petals-launcher module which is the container launcher.
- **petals-jbi**: Contains the JBI API code.
- **petals-jbi-descriptor**: Contains generated classes (from XML schemas) used to manipulate the JBI descriptors (components, service units, service assemblies, shared libraries ones).
- **petals-jbi-ext**: The JBI API extensions. This module contains some additional classes based on the JBI specification which are not available on the initial JBI API.
- **petals-jmx**: This module is a JMX client library implementation which can be used to manage the PEtALS container. This module is a good alternative to avoid complex JMX code since JMX client code is not maintainable at all…
- **petals-jmx-api**: The JMX client API.
- **petals-plugins**: This parent module contains the Maven (http://maven.apache.org) plugin used to build and package JBI artefacts from the source code. It also contains some archetypes (sort of patterns) used by Maven to create JBI artefacts source code projects.
- **petals-security**: This parent module contains generic security modules which are used by the kernel but which can also be used in other projects.
- **petals-shared-libraries**: This parent module contains some shared libraries which can be used by JBI components. More details on the shared libraries are available on the JBI specification.
- **petals-tools**: The parent module contains tools such as the administration and monitoring WebConsole, specific JBI components (JBI API exposed in JMX), installer, deployer, …
- **petals-ws**: This parent module contains Web Service related libraries such as WS-Notification, WS-Addressing. All of these libraries are based on the Web Service specification provided by the OASIS working group.

Distributions

The PEtALS container is distributed as ZIP archive. Once exploded, the folder structure is detailed below:

- **ant**: Contains Ant samples for JBI artefact (Component, Service Assembly or Shared Library) manipulation (installation, lifecycle management, …). This has to be used with the JBI Ant tasks provided with PEtALS.

- **bin**: Contains the scripts (*nix and Windows) needed to start, stop and get container status.
- **components** (quickstart only): Contains some of the PEtALS JBI components.
- **conf**: Contains all the container configuration files.
    - *server.properties*: Defines the local container properties such as name, timeouts, classloader behaviour, …
    - *topology.xml*: Defines the PEtALS network topology. This file contains all the information (IP addresses, ports, user/password, …) needed by containers to connect to each other plus some domain specific data.
    - *router.cfg*: Defines the router modules which are engaged.
    - *loggers.properties*: Defines the loggers levels of all the software components used within PEtALS. Note that each component logger level can be configured individually.
- **install**: This is the hot deployment folder. Each JBI artefact (Component, Service Assembly or Shared Library) which is dropped here is automatically deployed and started by the container.
- **installed**: All the installed JBI artefacts (Component, Service Assembly or Shared Library) are saved in this folder. When an an artefact is deleted from this folder, it is automatically stopped and undeployed by the container.
- **lib**: Contains the Java archives (Third party and PEtALS ones) needed by the PEtALS container.
- **licenses**: Contains all the third party library licenses.
- **repository**: Contains all the exploded JBI artefacts (Component, Service Assembly or Shared Library). Each installed artefact has a dedicated subfolder which is used as workspace at runtime to build component classloaders, store resources, … This folder is managed by the container, so please do not manipulate its resources.
- **schema**: Contains some XML schéma used to validate JBIi artefact configuration files.
- **uninstalled**: Contains all the JBI artefacts which have been successfully been uninstalled.
- **usecases** (quickstart only): Contains simple usecases used to introduce PEtALS and JBI usage.
- **webapps**: Contains some web applications (WAR files) which are deployed by PEtALS into the embeded JBI container. This folder is actually only used in the quickstart distribution to embed the monitoring and managemet web GUI.
- **work**: This folder is a temporary artefact folder used by the container during the installation and deployment phases.

Platform

The Platform distribution is the historical and the most advanced one. This distribution provides the distributed aspects introduced in the previous sections.

Standalone

The Standalone distribution goal is to run alone. Unlike the Platform distribution, there is no distributed registry (a simple hashtable replaces it), nor inter node message transport layer (replaced by an in-memory message transport layer).

This distribution is used when the distributed aspect is not the main goal of the infrastructure. The advantage is that all the distributed notions which are quite complex are removed from the container. The container is lighter, more robust and the response time (container startup, endpoint lookup, message transmission) is better.

Quickstart

The quickstart distribution is dedicated to new PEtALS users who want to discover what an ESB can provide. It is based on the standalone distribution and embeds the monitoring and management web GUI which is started by the container itself (Note that it is the Web console which is embeded in the container and not the opposite).

This distribution also comes with a detailed user guide and some use cases based on PEtALS components which help the user to understand the ESB benefits.

PEtALS JBI Components

All the PEtALS JBI components (Binding Components and Service Engines) are built upon the Component Development Kit.

The Binding Component collection is composed of:

- The **EJB component** (Enterprise Java Beans) exposes EJBs hosted on any EJB container as JBI service.
- The **SOAP component** is used to expose Web Services as JBI Services or to expose JBI Services as Web Services.
- The **File Transfer component** allows transfering files between JBI Services:
    - When a file is detected in a folder, it is send to a JBI Service.
    - When a message is received on a File activated JBI service, the message and its attachments are written into files.
- The **FTP component** (File Transfer Protocol) has the same feature than the File Transfer component but works with FTP servers and not with a local file system.
- The **JMS component** (Java Messaging Service) is used to publish/receive messages to/from JMS queues:
    - A JBI message received on the JBI service is translated to a JMS message and posted to the JMS Queue.
    - A JMS message posted to a JMS queue is detected by the component, then

translated to a JBI message and sent to a JBI service.

- The **Mail component** is used to communicate with mail server with the IMAP/POP/SMTP protocols.
    - A JBI message sent to a Mail activated endpoint is translated to a mail and sent to the mail server with the SMTP protocol. As a result a mail is sent by the mail server according to the recipient address.
    - The JBI component detects a mail on a mail server with the POP or IMAP protocol. Once the mail is translated to a JBI message, it is transfered to a JBI service.
- The **XMPP component** (eXtensible Messaging and Presence Protocol) is used to send/receive messages to/from a Jabber server.
    - A XMPP activated JBI service translates the incoming JBI message and send it to a Jabber server using the XMPP protocol.
    - When a message is received by the XMPP component (acts as a Jabber client), it is translated to a JBI message and sent to a JBI service.
- The **XQuare component** is used to access to databases. It uses XQuery to query a database through JDBC. Due to some XQuare limitations, an additional mode with native SQL has been added.

The PEtALS Service Engines are:

- The **BPEL engine** (Business Process Execution Language) is used to orchestrate JBI services. It is actually based on the OW2 Orchestra BPEL engine.
- The **CSV engine** (Coma Separated Values) is used to transform XML message to/from CSV format.
- The **EIP engine** (Enterprise Integration Patterns) is used for lightweight orchestration. Details about patterns can be found in (http://www.enterpriseintegrationpatterns.com). Some of the patterns provided by this component are:
    - The router to route messages to JBI service based on the message content (also named CBR for Content Based Router)
    - The splitter splits XML message into parts and send these messages to different JBI services.
    - The aggregator aggregates the results from multiple JBI service calls into a single message
- The **JSR181 engine** is used to expose JAXWS annotated Java classes as JBI services.
- The **Quartz engine** launches activities based on a crontab definition.
- The **validation engine** validates the content of the JBI message using a XSD

definition.

- The **XSLT engine** transforms JBI messages using an XSLT processor and XSLT files.

Some additional components such as a rules (based on JBOSS Drools), script (Groovy) ones are still under development.

PEtALS Tools

PEtALS is not only a JBI container and some JBI components. Various tools have been developed around this project in order to facilitate the users and developers life. This section describes the main ones.

PEtALS WebConsole

The PEtALS WebConsole is a Web GUI providing management and monitoring features for PEtALS containers.

The WebConsole uses the extended JMX API of the PEtALS container to:

- Collect data from all the containers of the management domain, processes this data and display messages content and useful statistics like service response time, service and container load, queue sizes, message repartition…
- Manage the JBI artefacts. It is possible to start, stop, deploy, and undeploy artefacts from the Web browser and to check their states.

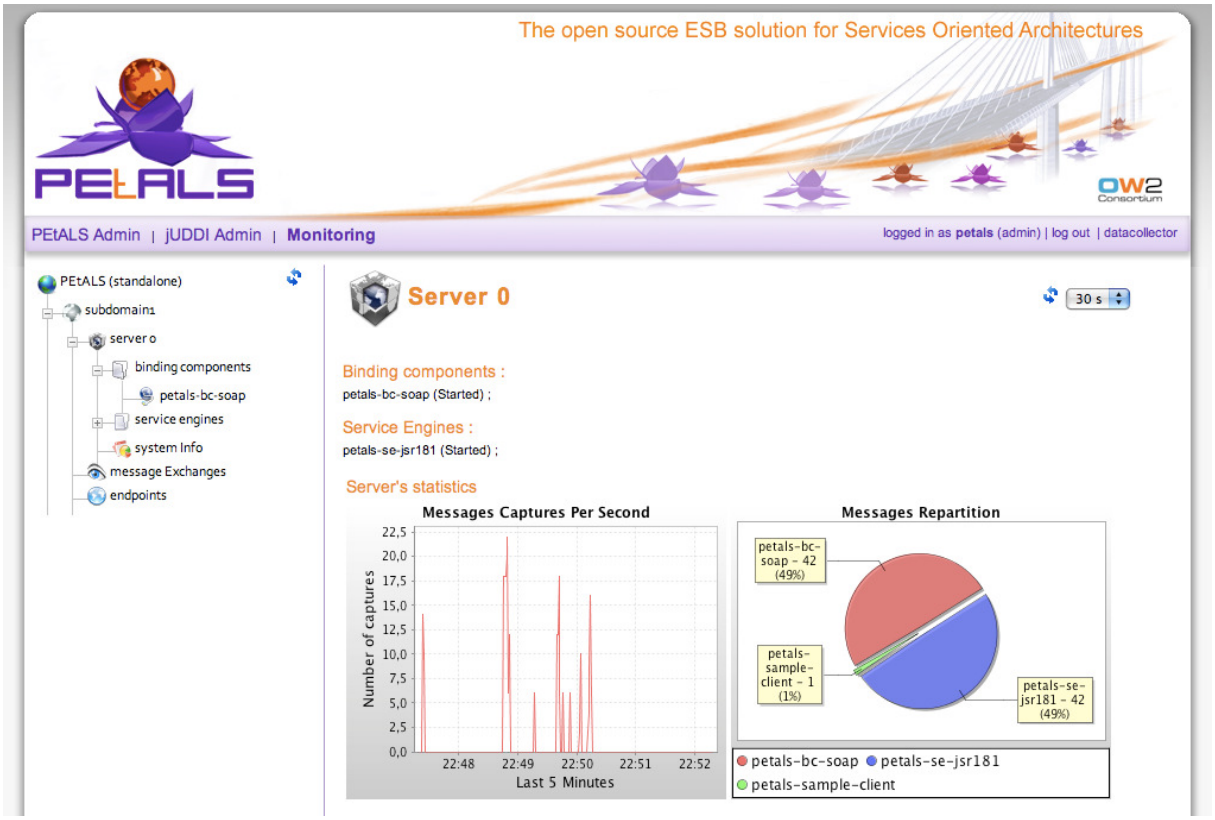

Figure 15 - WebConsole Monitoring

The WebConsole also provides:

- an embeded JBI client to invoke JBI services directly from the Web browser
- a module to configure services (Service Unit and Service Assembly generation) on the server side without any generation on the client side

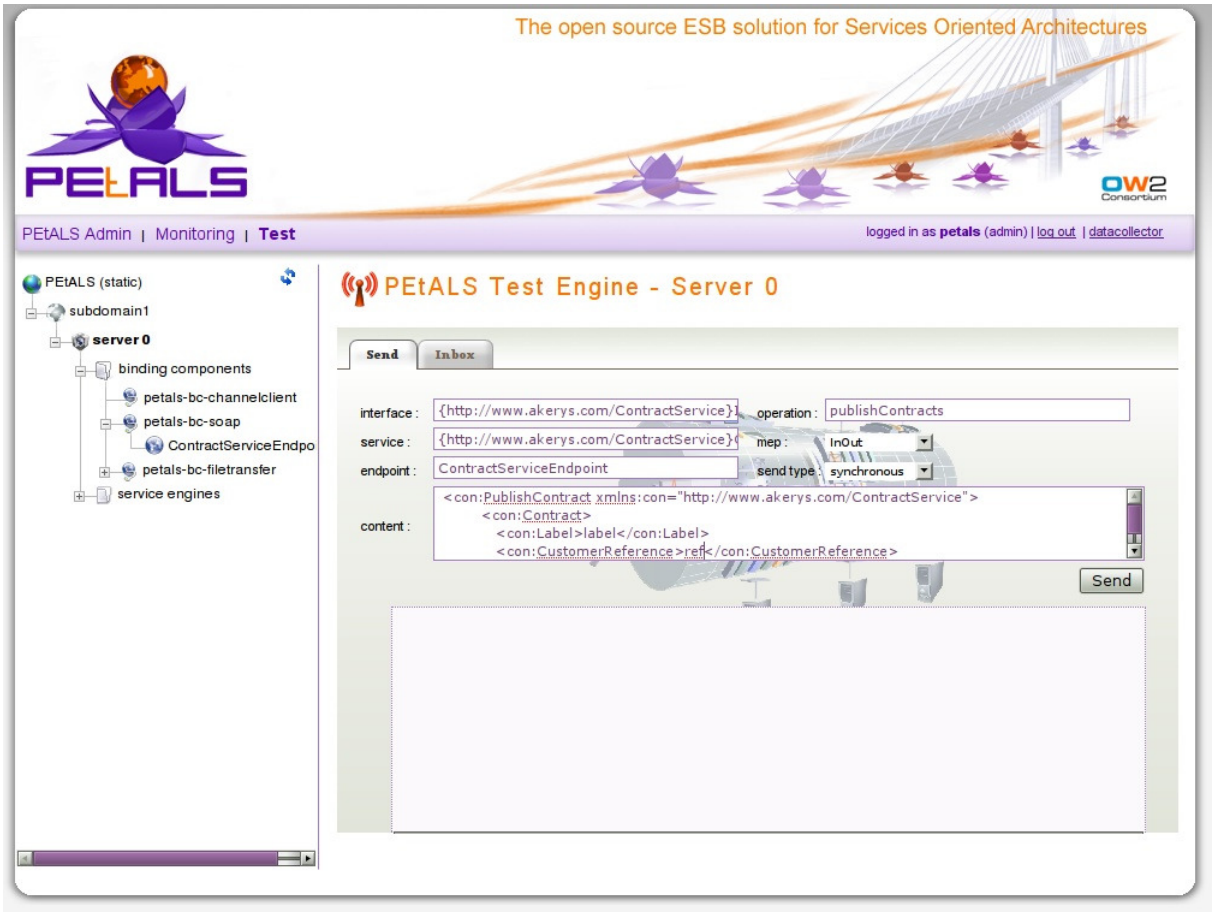

Figure 16 - WebConsole Embedded Client

High Level Architecture

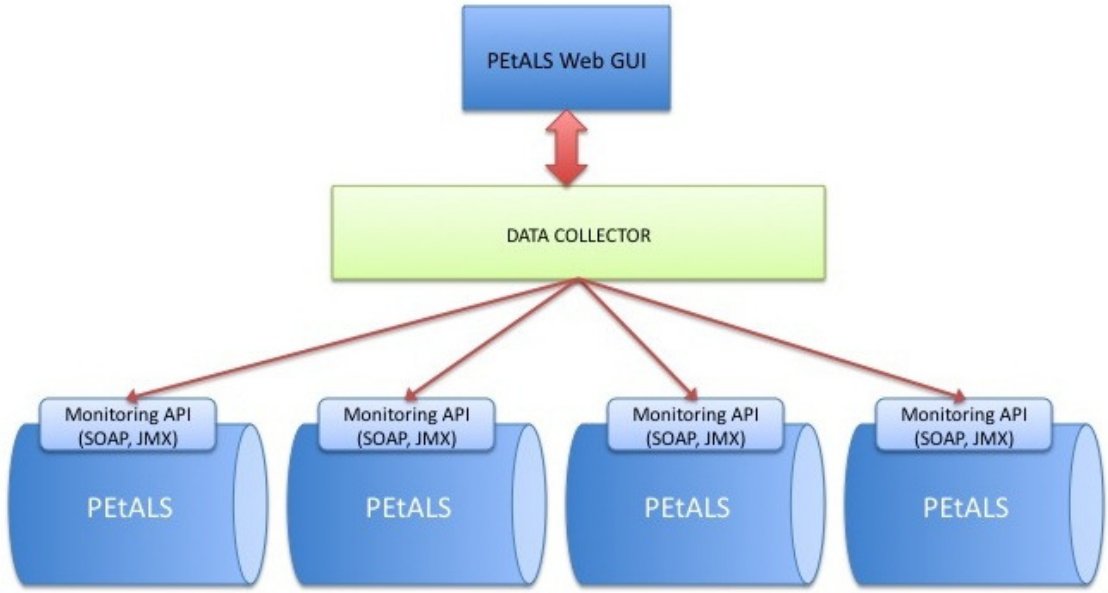

Figure 17 - WebConsole Architecture

The WebConsole is a single access point to manage containers distributed over several domains. The Web Application hosted on any Web Application server is connected to an intermediate layer (the data collector) which is in charge of establishing connections to the monitoring and management PEtALS container API.

A command (monitoring or management) sent to a container from the client (the web application) transits through the data collector layer and is routed to the right container. The actual architecture is based on the JMX technology. All the communication between or the parties uses JMX to exchange commands and datas.

Eclipse Plugins

Eclipse plugins provides advanced solution for developers:

- Create Service Units for the PEtALS JBI components with a set of wizard pages (No more XML file to edit). It is also used to package the SUs into SAs in order to be deployed directly in PEtALS. The wizards are automatically generated from the JBI component description files (XSD files).
- Create JBI component skeletons. This will help the component developer to start new projects based on the PEtALS CDK directly from the Eclipse IDE.

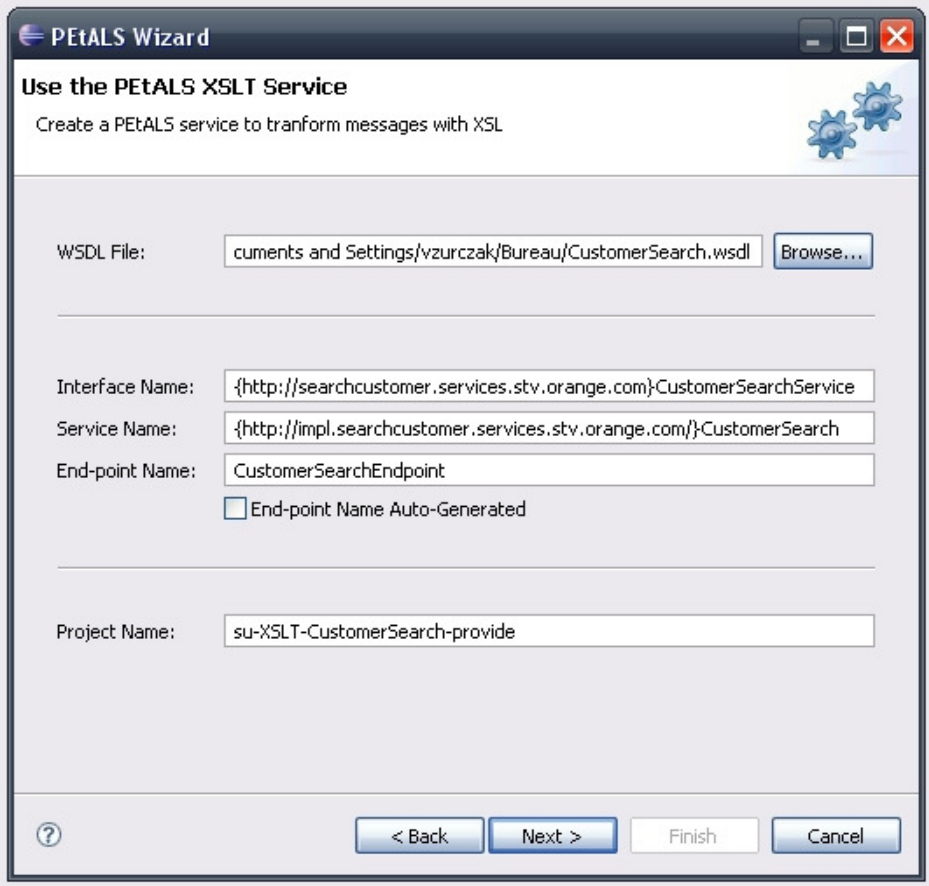

Figure 18 - Create a Service Unit with Eclipse Plugin

Note: Future work on the Eclipse plugins focuses on the containers monitoring and management integration.

Annexe A – Fractal Architecture

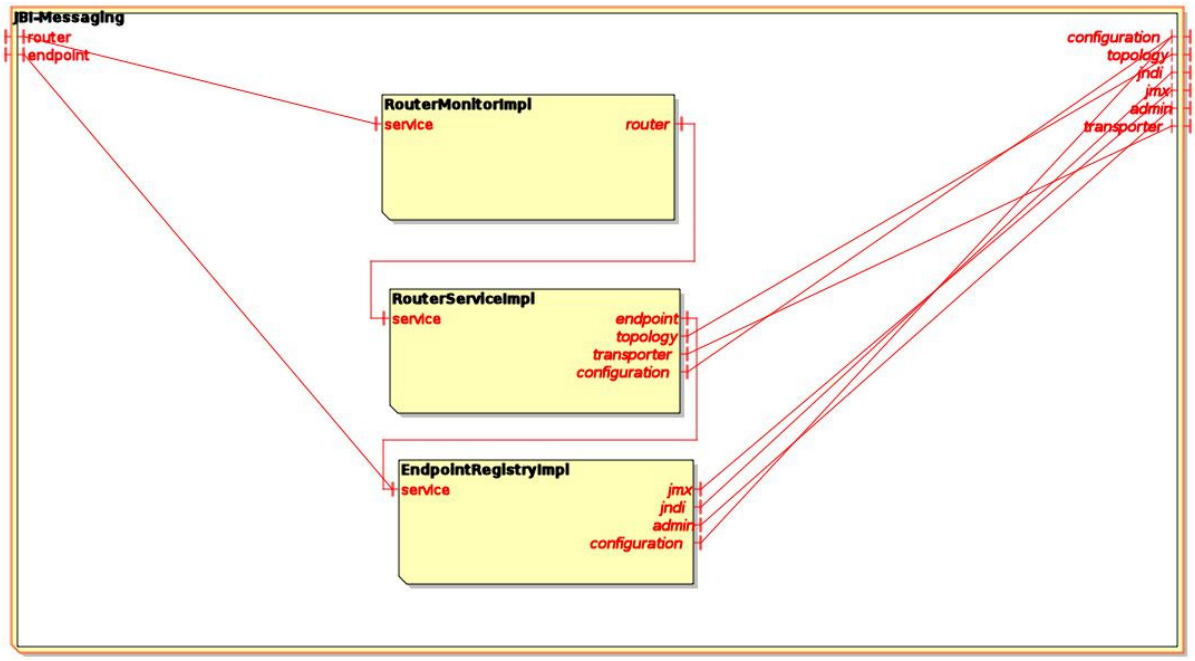

Figure 19 - JBI Messaging Composite

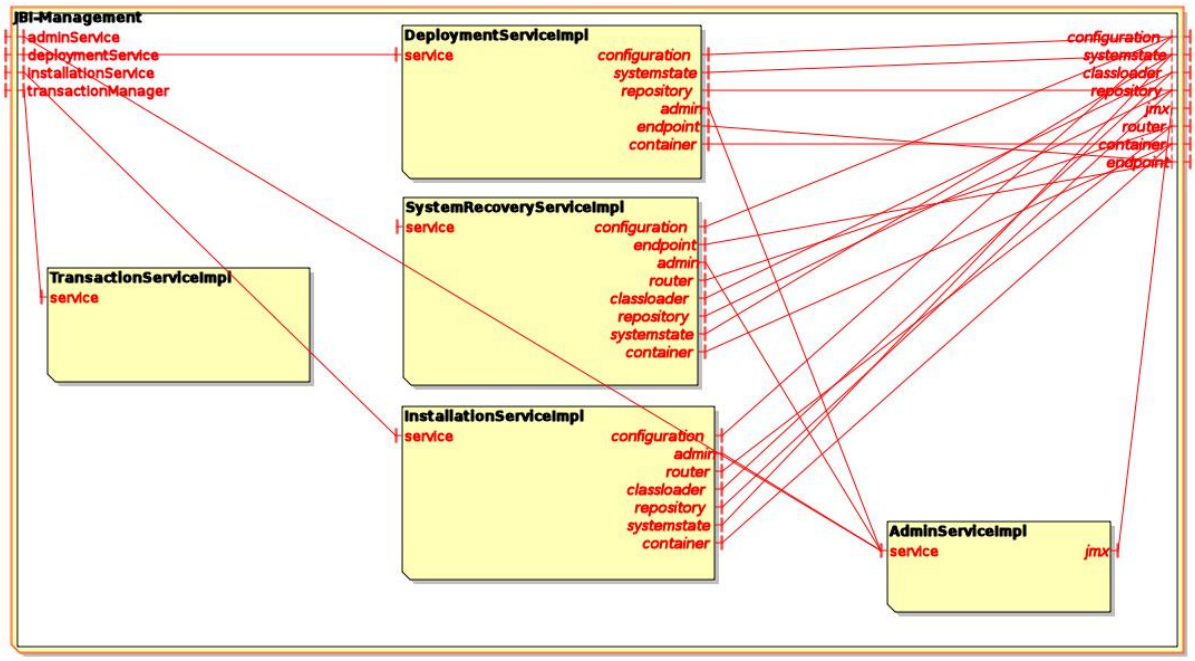

Figure 20 - JBI Management Composite

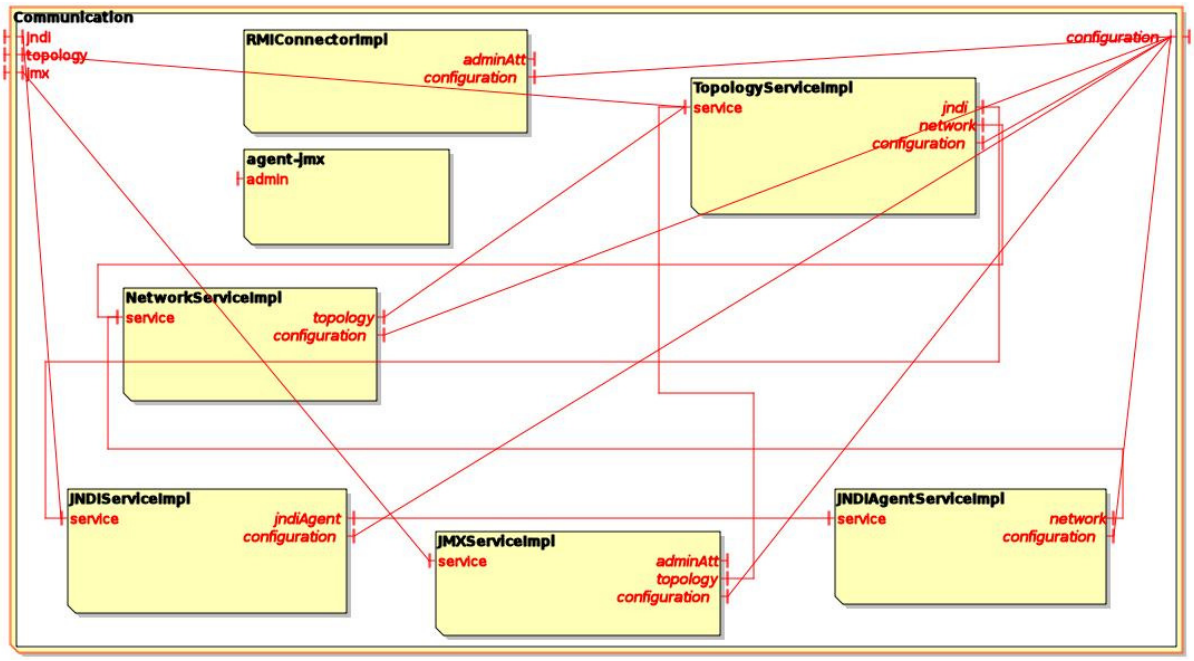

Figure 21 - Communication Composite

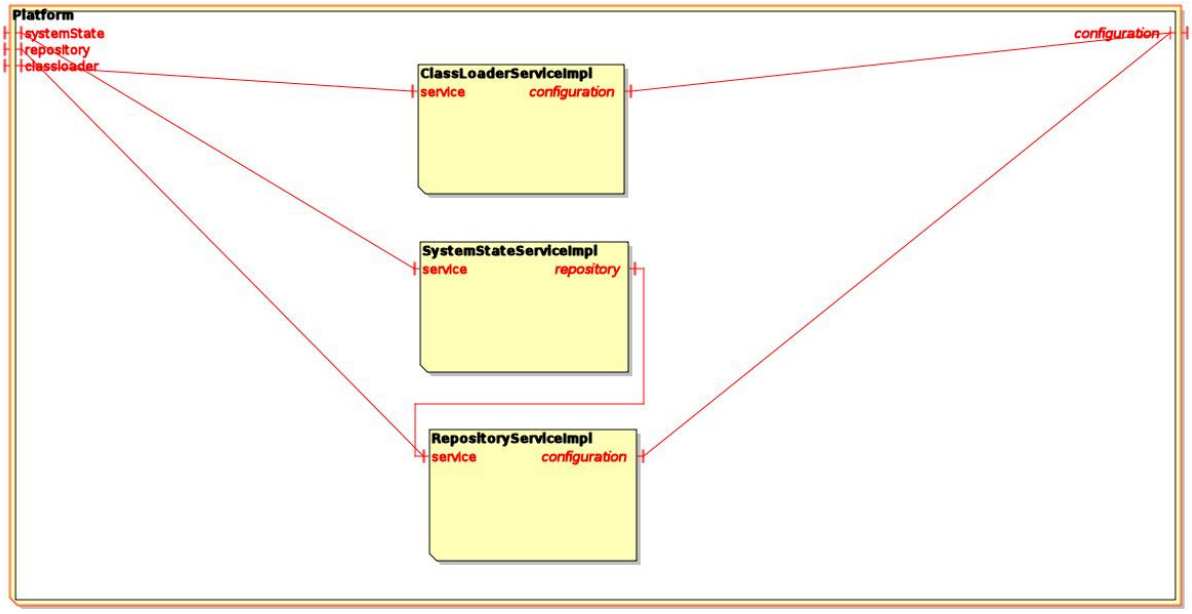

Figure 22 - Platform Composite

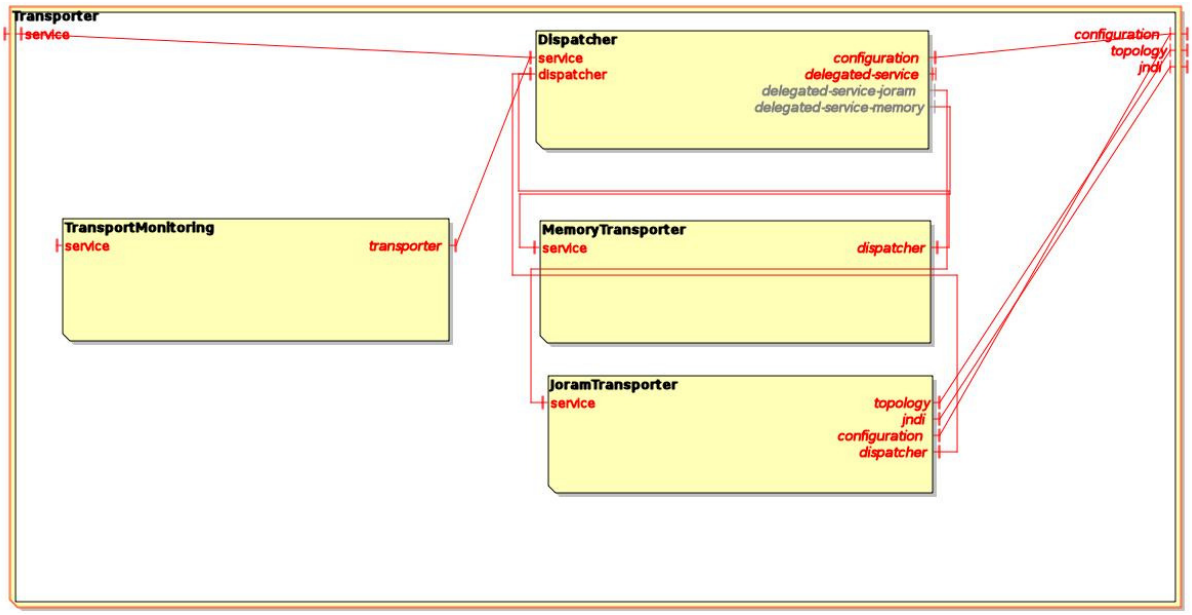

Figure 23 - Transporter Composite