

JONES Architecture

JONES Deliverable

Editor : G. BLONDELLE, V. QUEMA

Authors : G. BLONDELLE, P. GARCIA, A. LOUIS, V. QUEMA, J.B. STEFANI

Contributors :

Classification : Public

Deliverable no. :

Reference : TR/L1/INRIA/2

Date : March 2008

© EBM Websourcing, ENSTIMAC, France Telecom R&D, INRIA, OpenWide, Scalagent

**Project funded by Agence Nationale de la Recherche (ANR),
under the auspices of the Réseau national de recherche et
d'innovation en technologies logicielles (RNTL)**

Contents

1	Introduction	2
2	The JBI architecture	3
2.1	The JBI Components	4
2.1.1	The JBI Component structure	5
2.1.2	Component interactions with the container	6
2.1.3	Service consumer	7
2.1.4	Service provider	9
2.2	Configure Services	11
2.2.1	Service Assembly	11
2.2.2	Service Unit	12
2.2.3	Connection	14
2.3	The Normalized Message Router	15
3	The JONES environment	16
3.1	Current PETALS architecture	16
3.2	JBI composite component	17
3.3	FRACTAL components relationships	19
3.3.1	Messaging	19
3.3.2	Administration	20
4	The JONES distributed message router	21
4.1	Introduction	21
4.1.1	Targeted FRACTAL components	21
4.1.2	Final architecture	21
4.2	Towards an Extended Router	22
4.2.1	The Dream Transporter	22
4.2.2	Distant Communications with multiple transporters	23
4.2.3	The JBI Router as Transporter factory	23
4.3	Middleware Solutions	24
4.3.1	JORAM	25
4.3.2	DREAM	26

1 Introduction

This document presents the JOnES architecture. The overall objective of the JOnES project is to design and implement a distributed extension of the JSR 208 JBI specification [4]. Indeed, the current JBI specification describes an environment for installing, deploying and managing components (so-called *service engines*, *binding components*, and *service units*), that are assumed to be local to a given Java virtual machine (JVM). Extending the JBI specification to the distributed case is explicitly identified as future work in Chapter 14 of [4], that deals with “Future Directions” of the specification even if it is clear that distribution won’t be part of the JBI 2.0 standard.

Several motivations exist for constructing a distributed JBI environment. Most of them are related to classical benefits of distributed systems, in particular: ability to make use of parallelism and concurrency using multiple machines, thus increasing performance and availability; ability to place functions in selected locations and environments, thus increasing performance and security.

Extending the JBI specification to a distributed environment requires: (i) that routing functions provided by the message router at the core of a JBI environment be extended to deal with a network of JBI sites; (ii) that management functions identified in the specification now deal with multiple JBI sites, and provide the ability to set up and configure multiple JBI sites in the first place; (iii) that additional management functions be introduced to deal with specific distributed aspects, such as load sharing and performability management. This first version of the JOnES architecture document focuses on PETAALS, the first open-source distributed implementation of JBI, and routing extensions.

The design of a distributed JBI environment advocated in this document relies on an architecture-based approach, which can be summarized as follows:

- Develop a distributed JBI environment, including its message router, as a set of interconnected software components, with explicit and run-time control over component interconnection and component containment relationships.
- Use the explicit software architecture of a JBI environment as the primary input for its deployment, configuration, and management.

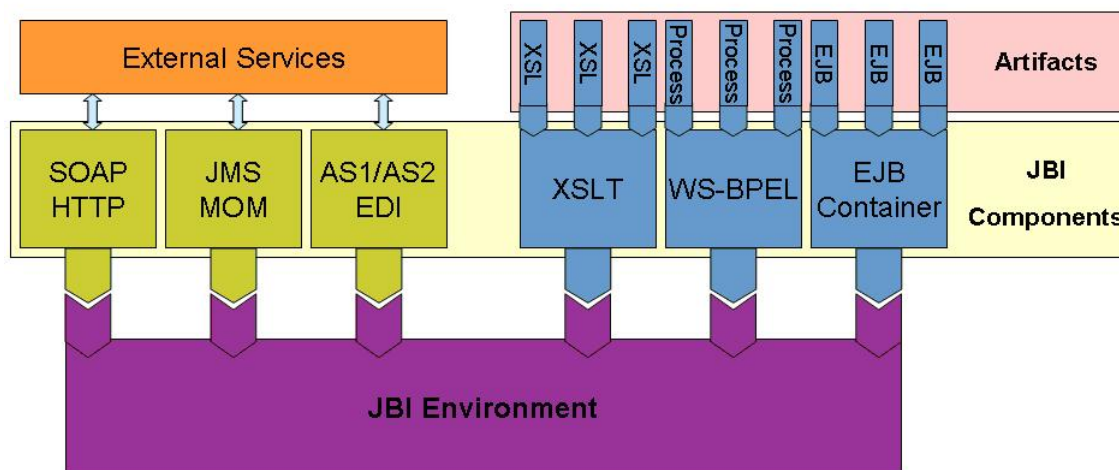
Specifically, this document advocates the choice of the FRACTAL component model as a basis of an architecture-based approach to the design of a distributed JBI. It is used for the construction of a JBI compliant local environment, and for the construction of a JBI compliant message router. Using FRACTAL for the construction of a local environment, brings several major benefits : (i) an explicit design of JBI architecture by composing software bricks: components (matching JBI entities); (ii) availability of components framework concerning different fields : messaging, deployment. . . (iii) deployment and configuration can be done in a distributed fashion, possibly involving synchronized deployment and configuration at multiple sites.

The document is organized as follows. Section 2 summarizes the main elements of the JBI architecture, as defined in the specification [4]. Section 3 describes the structure of a local JOnES environment. A single site JOnES environment provides a compliant implementation of a JBI environment. Section 4 describes the structure of a distributed JOnES router. A single site JOnES router configuration provides a compliant implementation of a JBI Normalized Message Router.


2 The JBI architecture

The high-level JBI architecture is described in chapter 4 of [4], and depicted by Figure 1 inspired by the section 4.3 of the JBI specification [4]. It comprises the following elements :

- *Plug-in components*, that can be essentially of two kinds: service engines, that provide high-level services (so-called *business logic*), and binding components, that provide connectivity between services. Service engines may in addition act as containers for other forms of components, called service units, which can be added and removed from their containers. Plug-in components and service units provide services, described using WSDL, and accessible at specific locations, called endpoints.
- *The Normalized Message Router*, which allows plug-in components to interact through *delivery channels* (each plug-in component has one delivery channel associated with it), thus providing access to services, with the mediation of binding components. Communication between a plug-in component and the normalized message router is in the form of so-called normalized messages, which typically comprise three parts: a payload, which consists of an XML document that conforms to a WSDL message type, without protocol encoding or formatting; metadata or message properties, that hold additional data associated with the message, and typically used during the handling of the message; a set of message attachments, referenced by the payload, which can be non-XML data.
- *Management functions* that support: the installation, deployment and configuration of plug-in components, of service units, and of groups of service units (or *composite assemblies*); the management of the life-cycle of plug-in components, service units, and composite assemblies; the query of execution information (called *component context*) and metadata of components and service providers in the JBI system.



Two types of JBI components:

 **Service Engines:** provide and consume business logic and transformation services

 **Binding Components:** provide connectivity to services external to a JBI installation

Figure 1: JBI container

2.1 The JBI Components

The Java Business Integration specification (Java Specification Request 208) defines a standard means for assembling integration components to create integration solutions that enable a service-oriented architecture (SOA) in an enterprise information system. These 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. A component is the main detail a user or developer faces when working in the JBI environment. Understanding how a component interacts with the JBI environment is just as important as understanding the environment black-box itself.

JBI defines a container that can host components. Two kinds of components can be plugged into a JBI environment:

- **Service engines** provide logic in the environment, such as XSL (Extensible Stylesheet Language) transformation or BPEL (Business Process Execution Language) orchestration.
- **Binding components** are sort of "connectors" to external services or applications. They allow communication with various protocols, such as SOAP, Java Message Service, or ebXML.

JBI is built on top of state-of-the-art SOA standards: service definitions are described in WSDL (Web Services Description Language), and components exchange XML messages in a document-oriented-model way.

The JBI environment glues together the JBI components by acting as a message router to:

- Find a service provided by a component
- Send a request to a service provider
- Manage the message exchange lifecycle (request, response, acknowledgements, etc.)

It also provides a set of services (support of naming context, transactional context, etc.) to the components.

In addition, the JBI container manages, through a rich management API, the installation and lifecycle management of the components, and the deployment of artifacts to configure an installed component (for instance, deployment of XSL stylesheets to a transformation service engine).

2.1.1 The JBI Component structure

The JBI components are the base elements composed by the JBI container to create an integration solution. The components are plug-ins for the container and are considered "external." As a Java EE (Java EE is Sun's new name for J2EE) container hosts Enterprise JavaBeans (EJB) components, or as a portal hosts portlets, the JBI container hosts integration components. You can write your own components, or you can use components written by someone else, in the same way you write a portlet or use an existing one.

A JBI component comes with several main objects represented by the following SPI (system programming interface):

- The *Component* object, with which the container interacts to retrieve component information (such as a services description)
- The *LifeCycle* object, used by the container to manage the component's lifecycle
- The *ComponentContext*, given by the container to the component, for communication with the JBI environment

Additional objects are:

- The *Bootstrap* object, which provides all operations required at install/uninstall time
- The *ServiceUnitManager* object, used to manage deployment of artifacts on a component

A component is packaged as an archive (a zip or jar file). This archive contains the component's classes, the required libraries, and a descriptor file.

To plug a component into a JBI container, use the management API provided by the container. This API allows you to provide to the container your component package's location. Then, the container processes the component archive and installs the component.

From the component viewpoint, two different phases are defined:

- The *installation phase*, in which the JBI container installs the component and plugs it into the bus
- The *execution phase*, in which the JBI container interacts with the component

Installation During the installation phase, the component must perform all extra processes needed for its execution, such as the creation of mandatory folders or the installation of a database. As illustrated in figure 2, the *Bootstrap* object completes the installation and receives an *onInstall()* event from the container with an associated *InstallationContext* object, which provides to *Bootstrap* some installation information (the path of the installation, a naming context, etc.).

Execution The container starts, stops, and shuts down the component. As shown in Figure 3, the container initializes a component by passing it a *ComponentContext*, which is the entry point to the JBI environment. While the component runs, it interacts with the JBI environment through this *ComponentContext*.

The component can consume services (exposed on the bus by other components) by sending messages to a service provider. The component can also act as the service provider, accepting and processing such messages, and returning an answer to the consumer through the bus.

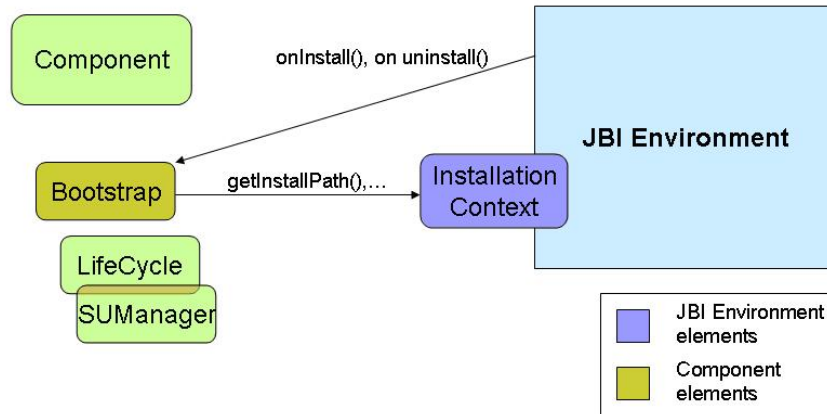


Figure 2: Interactions during the installation phase

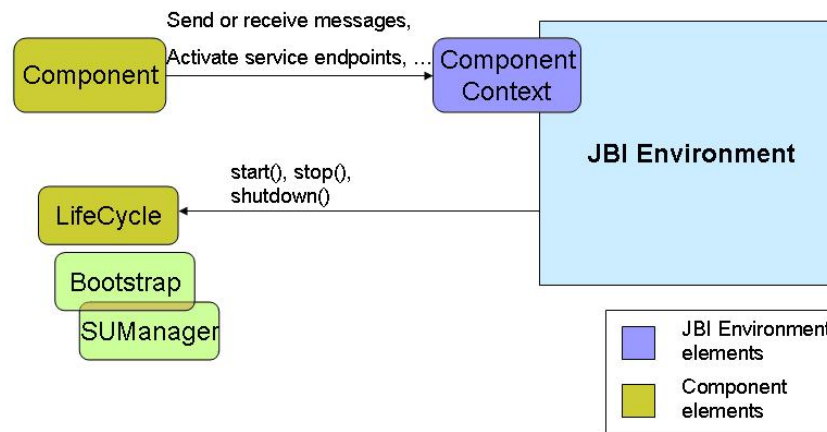


Figure 3: Interactions during the execution phase

2.1.2 Component interactions with the container

To illustrate the interactions between a service consumer and a service provider, let's take a simple request-response exchange as an example. The corresponding message exchange pattern (MEP) is an InOut exchange pattern, as defined in the JBI specification (see section 5.4.2 of the specification [4]). By default, JBI supports four WSDL pattern exchanges: In, InOut, InOptionalOut, and RobustIn. Each pattern defines a particular exchange sequence. Let's begin with service consumer interactions.

2.1.3 Service consumer

Once a component is running, it can find and consume the services registered in the JBI environment. Therefore, the component is in the role of a service consumer.

Find a service endpoint

A `ServiceEndpoint` represents an address where a service provided by a component can be found. Several components can provide the same service (with eventually different implementations), but each of those components has a unique endpoint. As shown in Figure 4, the consumer gets an endpoint matching a service name by calling `getEndpointsForService(serviceName)` on its `ComponentContext`. The consumer then chooses the provider it wants to reach. If the consumer already knows the address of the provider that it wants to contact, it can retrieve the provider `ServiceEndpoint` object by calling `getEndpoint(serviceName, endpointName)`.

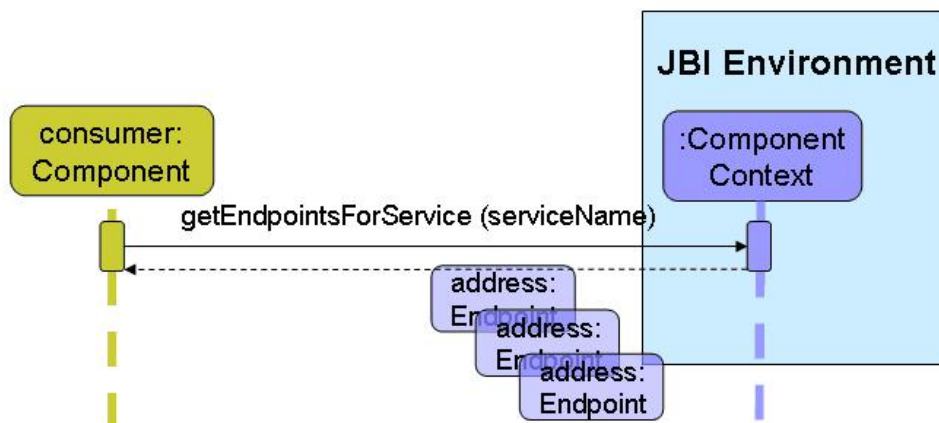


Figure 4: Get endpoints for a given service

Create a message exchange

To manipulate messages, the `ComponentContext` provides a `DeliveryChannel` object, which represents a bidirectional communication channel between the component and the JBI environment's message router (called *normalized message router*, or NMR). The `DeliveryChannel` is in charge of message-exchange instantiation and is the path through which the messages advance to the NMR. The NMR routes the message to the component that provides the requested service. An exchange between the consumer and the provider is materialized by a `MessageExchange` object. This object serves during the exchange's entire lifecycle.

When the consumer wants to initialize a new exchange, it asks a `MessageExchangeFactory` (provided by the `DeliveryChannel` object) to create a new `MessageExchange`. This `MessageExchange` contains the actual message content and a set of metadata, such as the provider endpoint, the status of the exchange (active, done, in error), or identification of the exchange "owner" (the consumer or the provider). The consumer and provider share `MessageExchange` during the exchange (request, response, etc.).

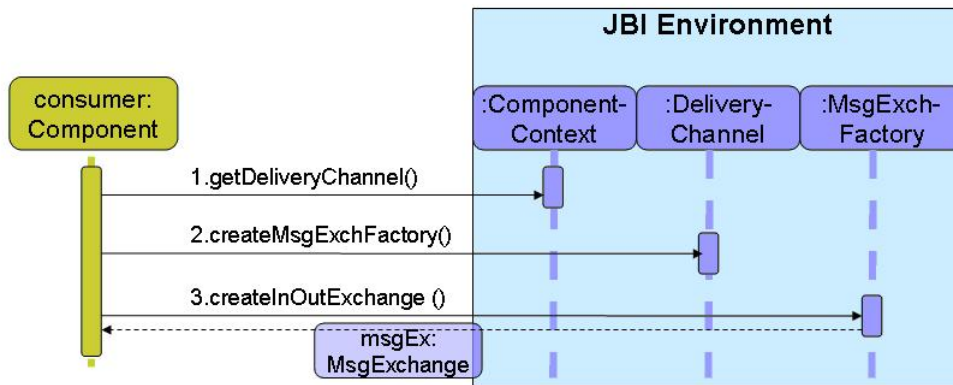


Figure 5: Create a message exchange

Send the message

Now that the consumer has instantiated a MessageExchange, it can set on this object the message it wants to send to the consumer. The consumer has to set a NormalizedMessage as the exchange's "in" message. The NormalizedMessage is a JBI definition of a message. The consumer asks the MessageExchange to create a new NormalizedMessage.

Then, the consumer sets on this NormalizedMessage the content of the message (an XML payload) and eventually some attachments. The consumer must set on the MessageExchange the provider's ServiceEndpoint (previously retrieved) and the name of the operation to be performed.

Note: The consumer can omit stipulation of the provider's ServiceEndpoint and just specify a service name. In this case, the NMR will search all matching endpoints and choose one of them.

Finally, the consumer sends the MessageExchange using the DeliveryChannel. Figure 6 illustrates this entire process.

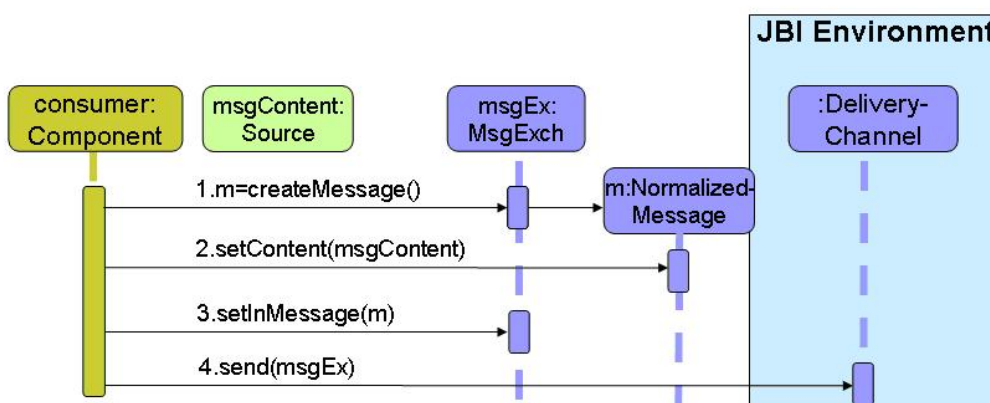


Figure 6: Send a message

2.1.4 Service provider

Once a component is running, it can also provide services. This component acts as a service provider.

Activate an endpoint

The provider must publish the services it wants to offer. As shown in Figure 7, the `ComponentContext`'s `activateEndpoint(serviceName, endpointName)` method publishes the service publication. This method returns a generated `ServiceEndpoint` object that references the new service in the JBI environment. With publication completed, other components can access the activated services by finding the corresponding endpoint with their `ComponentContext`'s `getEndpointForService()` method.

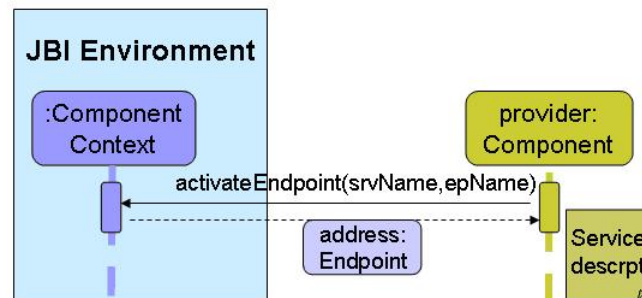


Figure 7: Activate a ServiceEndpoint

Receive a message

Now that the provider has published some services, it can receive messages from other components (consumers). When a consumer sends a message to a provider, the message (`MessageExchange`) is pushed in the message queue of the provider's `DeliveryChannel`. The provider retrieves the received messages from the message queue by calling `accept()` on its `DeliveryChannel`. Once the provider obtains a `MessageExchange`, it can process it. The provider gets the "in" message (the `NormalizedMessage` object set by the consumer), the name of the operation to perform, the payload of the message, and so on.

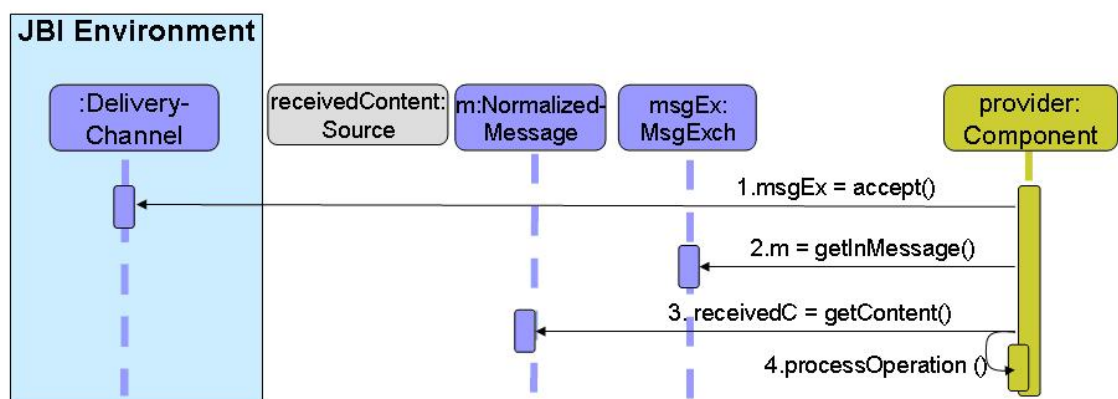


Figure 8: Receive a message

Send the response

If the operation requires an answer, the provider can set an "out" message on the MessageExchange and send it again to the NMR via its DeliveryChannel. The NMR routes the MessageExchange to the consumer that previously initiated this MessageExchange.

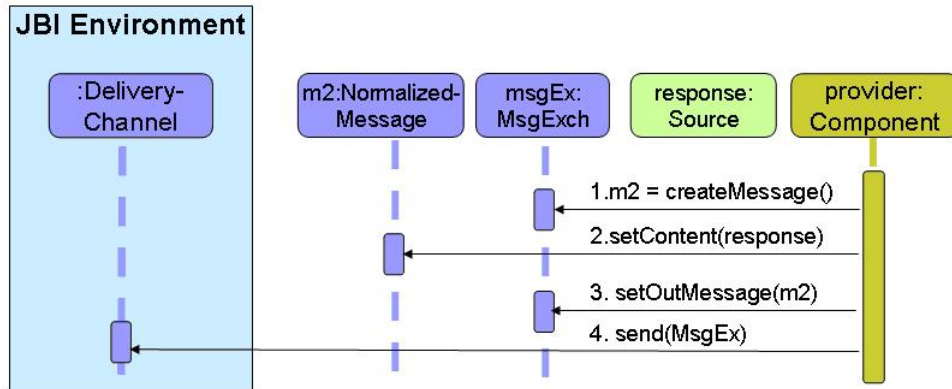


Figure 9: Send the response

Close the exchange

After the provider sends its response, the exchange is nearly complete. The NMR routes the answer to the consumer, which receives it with an `accept()` call on its `DeliveryChannel`. The consumer processes the `MessageExchange`'s "out" content and then must close the exchange. To do this, the consumer sets the status of `MessageExchange` to `DONE` and sends it again to the NMR via its `DeliveryChannel`'s `send()` method. The exchange is terminated, and the provider learns of this termination upon receiving the `MessageExchange` with its `DONE` status. The schema 10 describes the whole exchange process.

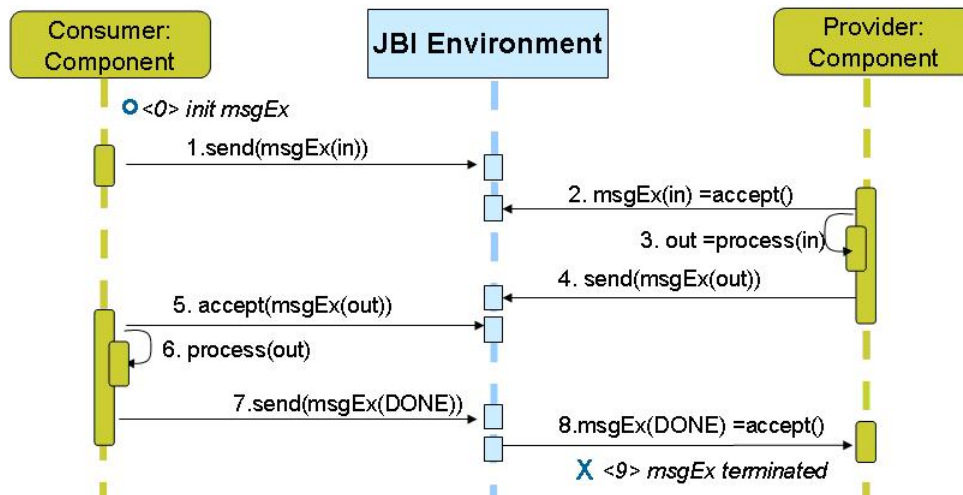


Figure 10: The whole in-out exchange

2.2 Configure Services

2.2.1 Service Assembly

A Service Assembly is a descriptor used to configure the components in order to integrate a global application.

A Service Assembly is deployed on the JBI container as a simple ZIP archive. The container analyses all the content in this archive and configure the impacted components. The configuration for a component is called a ServiceUnit. When we configure a Component with a Service Unit, we say that we deploy an artefact on it. Note that the content of a Service Unit is opaque to the container and is just provided to the Component.

If the Service Assembly contains links definitions between services (also called Connections), the container creates them. Connections can be seen as alias to real ServiceEndpoints.

The Service Assembly is a ZIP archive containing an XML descriptor file, which defines to which Component a Service Unit is provided, and the connections between services. The Service Assembly contains also each Service Unit to deploy to Components. These Service Units are also ZIP archives.

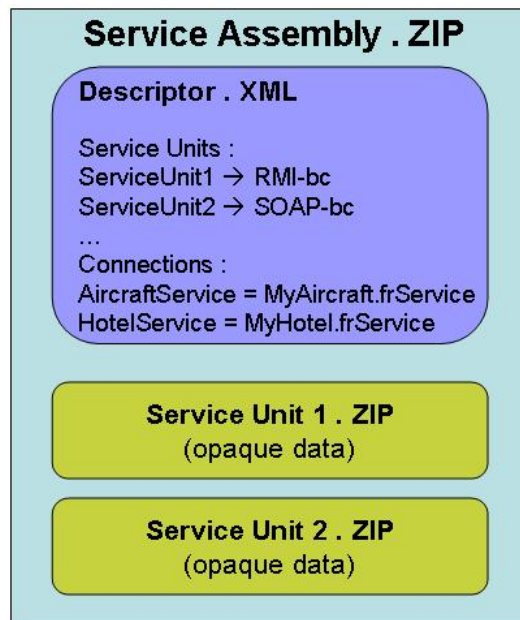


Figure 11: a Service Assembly

So, the role of a Service Assembly is to define the glue between a set of services in order to create an application, in a SOA way. As multiple Service Assemblies can be deployed on the container, multiple applications can run on it and share the same services.

2.2.2 Service Unit

A Service Unit is a set of elements deployed to a JBI Component. A Service Unit contains an XML descriptor that the Component can read, and any artefact that the Component can require. For instance, a Service Unit for an XSLT Engine contains an XSL style sheet, and the descriptor file tells to the Component that this style sheet has to be used when a message is received on a particular ServiceEndpoint.

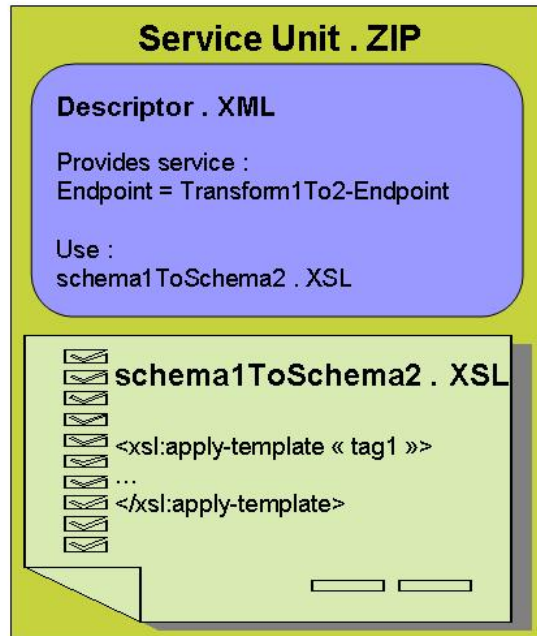


Figure 12: a Service Unit

The container explodes the ServiceUnit archive in a directory and calls the Component's Service Unit-Manager (which is a part of a JBI Component) to inform it that a new Service Unit is available, and indicate the directory where the Service Unit has been exploded. The Service Unit Manager of the Component registers this new Service in the JBI environment and configures the Component to use the related artefact (XSL style sheet...) with the registered ServiceEndpoint.

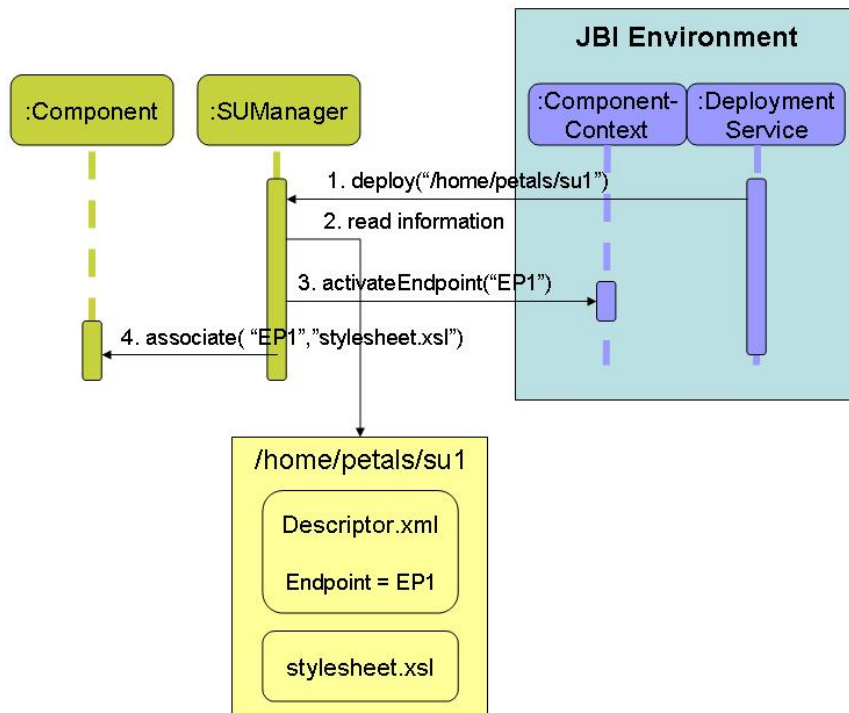


Figure 13: deploy a SU on a Component

When the Component receives a message for a particular ServiceEndpoint, it knows which artefact to use.

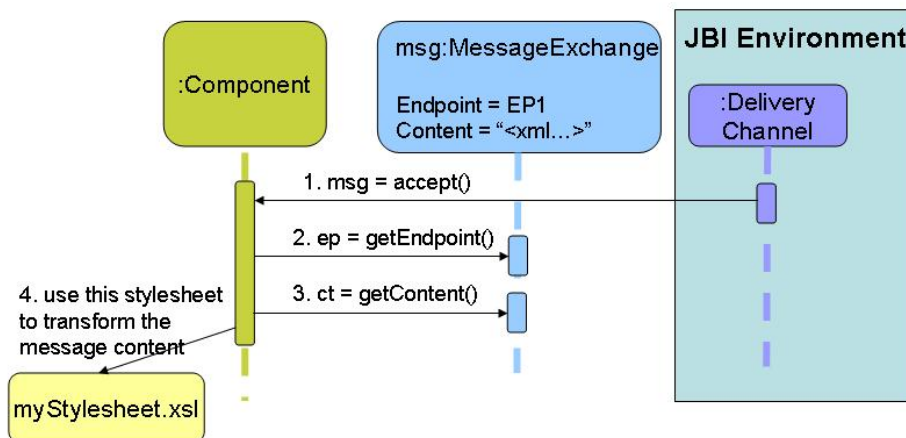


Figure 14: use artefact for particular ServiceEndpoint

2.2.3 Connection

A Connection is defined in the descriptor of a Service Assembly. It can be seen as an alias for a real ServiceEndpoint. So, a Component can specify “I want to send a message to the Service Endpoint called ‘My Endpoint’.”, and a connection specifies that “My Endpoint” is in fact “My Real Endpoint”. When the Component sends the message to “My Endpoint”, it is sent to “My RealEndpoint”.

The interest of using such a mechanism is to dynamically reconfigure the links between Components. For instance, a Component, such as a Travel-Agency-Workflow engine, is configured to call a service which ServiceEndpoint name is “Airline ServiceEndpoint”. Then, depending on the partnerships of the Travel Agency with the Airline Company, the connection will link the “Airline ServiceEndpoint” ServiceEndpoint with, for instance, “My Low Cost Airline Company Endpoint”, “Another Airline Company Endpoint”. There is no change to apply to the consumer of this service.

Just by deploying various Service Assemblies containing different connections, the application will dynamically change the services that are used.

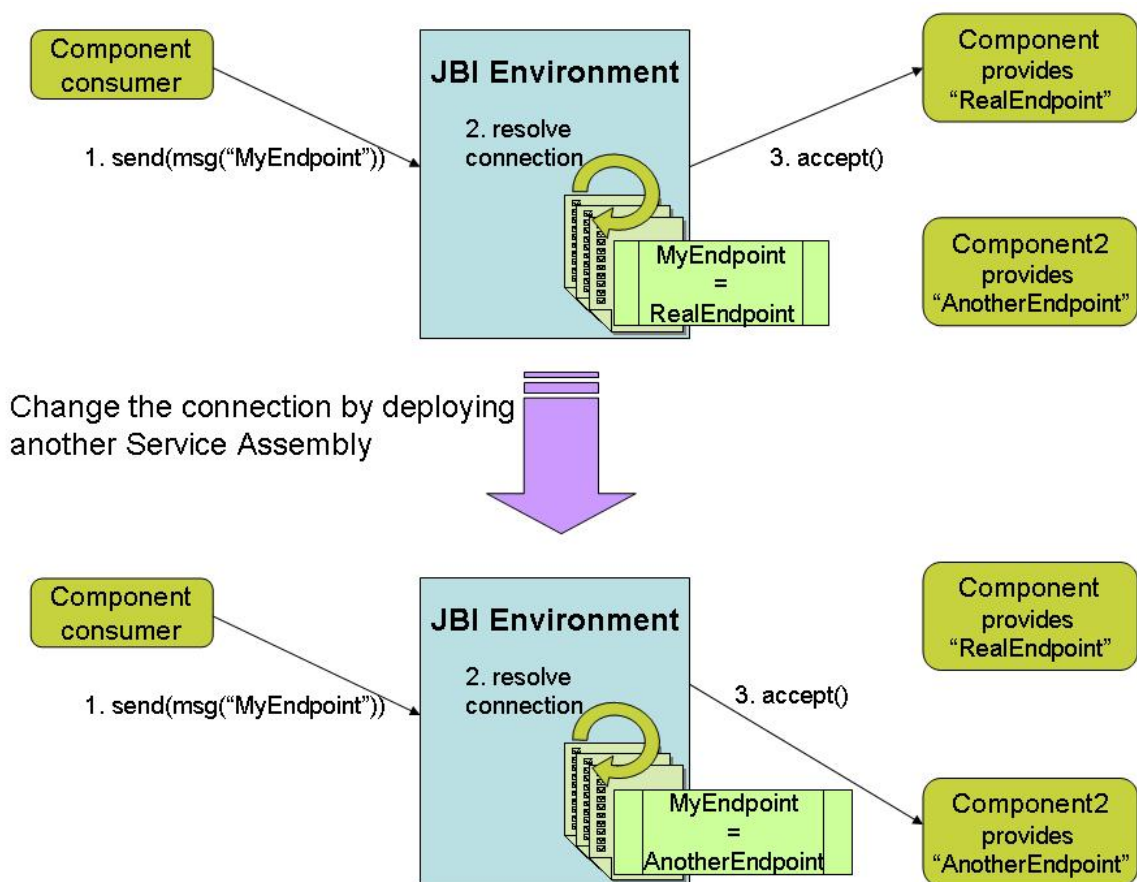


Figure 15: resolve a connection

2.3 The Normalized Message Router

The normalized message router, or NMR, receives message exchanges from JBI components (engines or bindings), and routes them to the appropriate component for processing. The normalized message router operates on message exchanges, which are containers for normalized messages, whose payload is an XML document. The mediated message-exchange processing model decouples service consumers from providers, and allows the NMR to perform additional processing during the lifetime of the message exchange.

The link between JBI component and Normalized Message Router is a synchronized bidirectional communication pipe named Delivery Channel. When a component sends a message, the NMR is in charge of finding the appropriated component that hosts the requested services, according to routing rules.

The JBI specification also defines some rules to route the message such as the connection mechanism that has been previously seen. The way to elect a specific service implementation, when multiple services match the request, is not discussed in the JBI specification.

The interactions through the normalized message router that are described by the JBI specifications are of four kinds :

- One way service requests
- Reliable one way service requests (one way with possible fault notification)
- Service request with response
- Service request with optional response

The NMR is presented, in the specification, as an element that sends messages between components that are hosted by a single container (as shown on the figure 16). One approach of the JOnES project is to extend the NMR and to allow it to be connected to other NMR from other JBI containers, spreading messages communication over a network of JBI container.

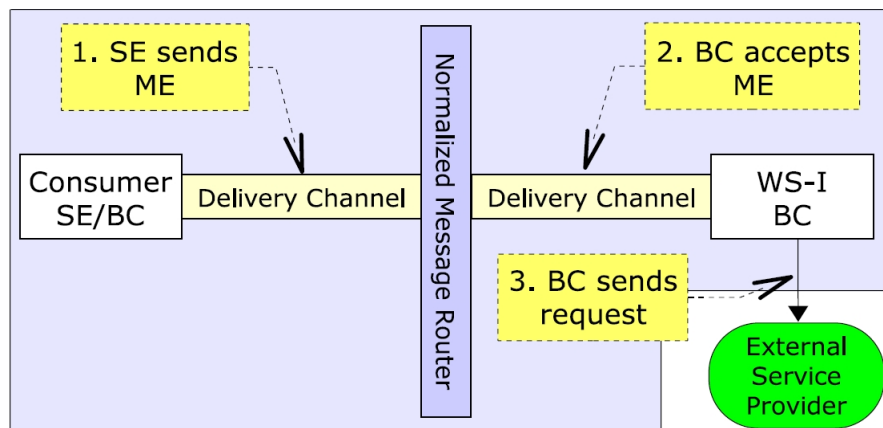


Figure 16: One way service request to an external provider

3 The JONES environment

3.1 Current PETALS architecture

The current JONES environment corresponds to a FRACTAL implementation of the JBI specification. This implementation is the PEtALS container [3] .

The following figure represents the PEtALS fractal architecture :

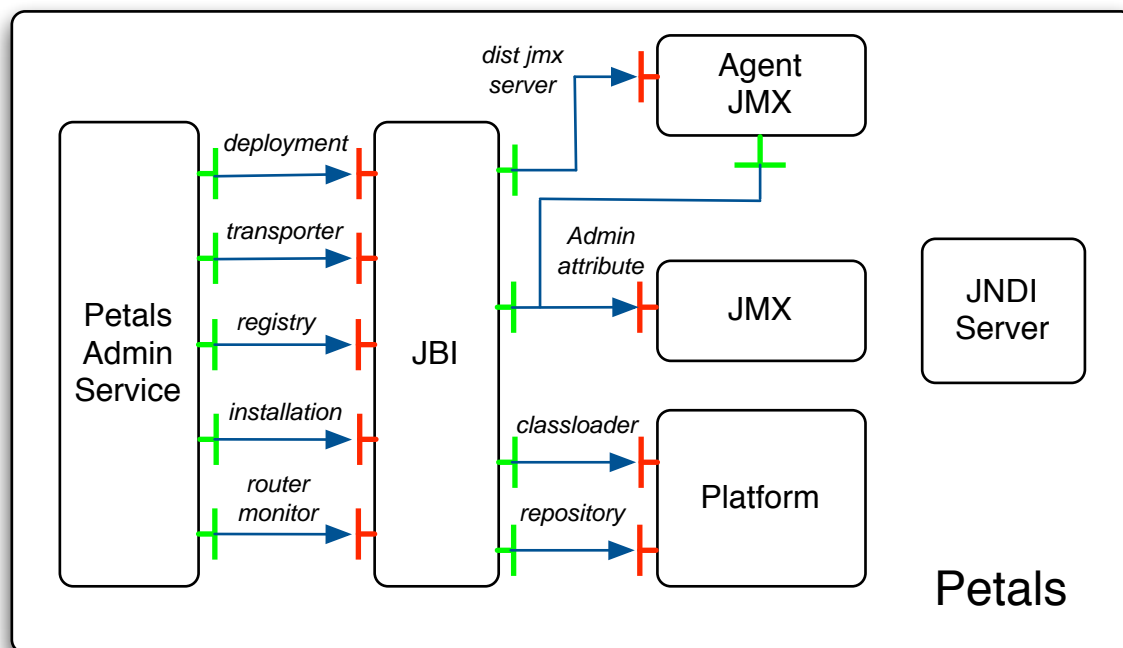


Figure 17: PEtALS fractal architecture

The PEtALS component includes the following FRACTAL components.

- Platform : this component contains two sub-components :
 - Repository: this component provides file system management facilities used by the JONES environment.
 - ClassLoader: this component provides the class loading facilities required by the JBI specification at installation time (bootstrap class loader), and at execution time (execution class loader).
- JBI : the main component detailed in section 3.2
- PetalsAdminService: this component provides container-administration functionalities and monitoring reporting.
- JNDI Server : this component is the base of the distributed communication for the Registry FRACTAL component. It manages distributed communication between containers for business services registration.

- Distributed JMX agent : this component allows communication with other containers, for administrative tasks. It strongly uses the JMX FRAGMENT component.
- JMX : this component is the base of the distributed communication for the Distributed JMX agent FRAGMENT component. It allows the JMX communication between two containers for internal management.

3.2 JBI composite component

The JBI component contains several FRAGMENT sub-components. This sub-section aims at describing main sub-components and their bindings.

Components related to routing

- EndpointService : this component provides access to Services description and localization.
- Registry : this component provides the way to access to the Services-localization directory. Each service provided by a component and other meta-data (containers information, servers topology ...) are registered through this Registry.
- Router : this component provides the logic to find the destination for a message, according to the required Service, localization, or other routing rules.
- Transporter : this component provides a message communication service within the JONES environment.
- RouterMonitor : this component wraps the Router component and catch various information for monitoring.

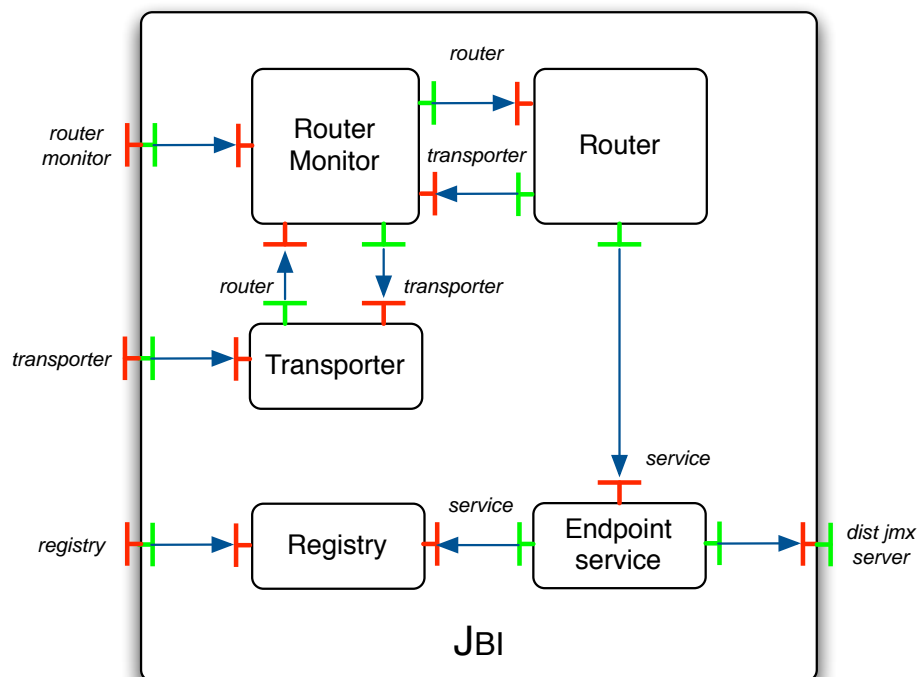


Figure 18: Components related to routing contained in the JBI component

Components related to administration

- Lifecycle Manager : this component provides the interfaces required for controlling the life cycle of JBI services and JBI plug-in components (binding components and service engines).
- JBI Services : these components provide interfaces required by a JBI environment:
 - Admin implements the `AdminServiceMBean` interface.
 - Installation implements the `InstallationServiceMBean` interface.
 - Deployment implements the `DeploymentServiceMBean` interface.
- AutoLoader : this component manages the hot (un)deployment of Components and ServiceAssemblies. The AutoLoader pilots the standard JBI administration to perform the deployment of those elements.
- SystemState : this component keep the state of the system. When the PETALS container stops or crashes, the SystemState has to restore, on the next start, JBI components or ServiceAssemblies according to their previous state.

The following FRACTAL scheme illustrates bindings between components matching the first two items.

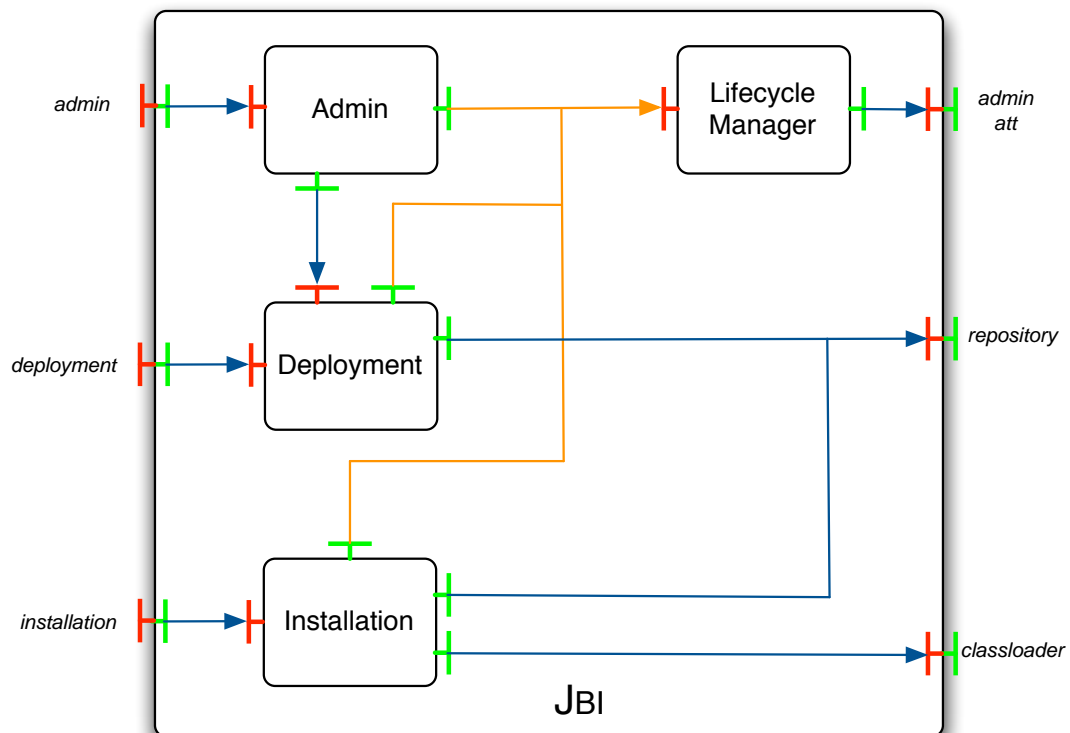


Figure 19: Components related to administration contained in the JBI component

3.3 FRACTAL components relationships

3.3.1 Messaging

The current PETALS internal messaging design is depicted on the figure 20. It relies on a distributed implementation of the NMR. It allows the functions of the normalized message router to be supported by multiple machines simultaneously, thus providing scope for increased performance and availability. The distant PETALS environments communicate via JORAM, an open source implementation of JMS (sub-section 4.3.1). The use of JORAM is transparent to JBI components and thus makes this implementation compliant with the JBI specification. The Transporter, FRACTAL component plotted on figure 18), is in charge of these communications. Besides, distant PETALS environments share information via a distributed JNDI directory (also called ServiceLocation directory).

When a JBI component sends a message via its DeliveryChannel, the message is forwarded to the Router FRACTAL component. The Router contacts the Registry to access the distributed ServiceLocation directory. It retrieves all the matching services. Then the Router elects one of these services. It finally delegates the transport of the message to the Transporter.

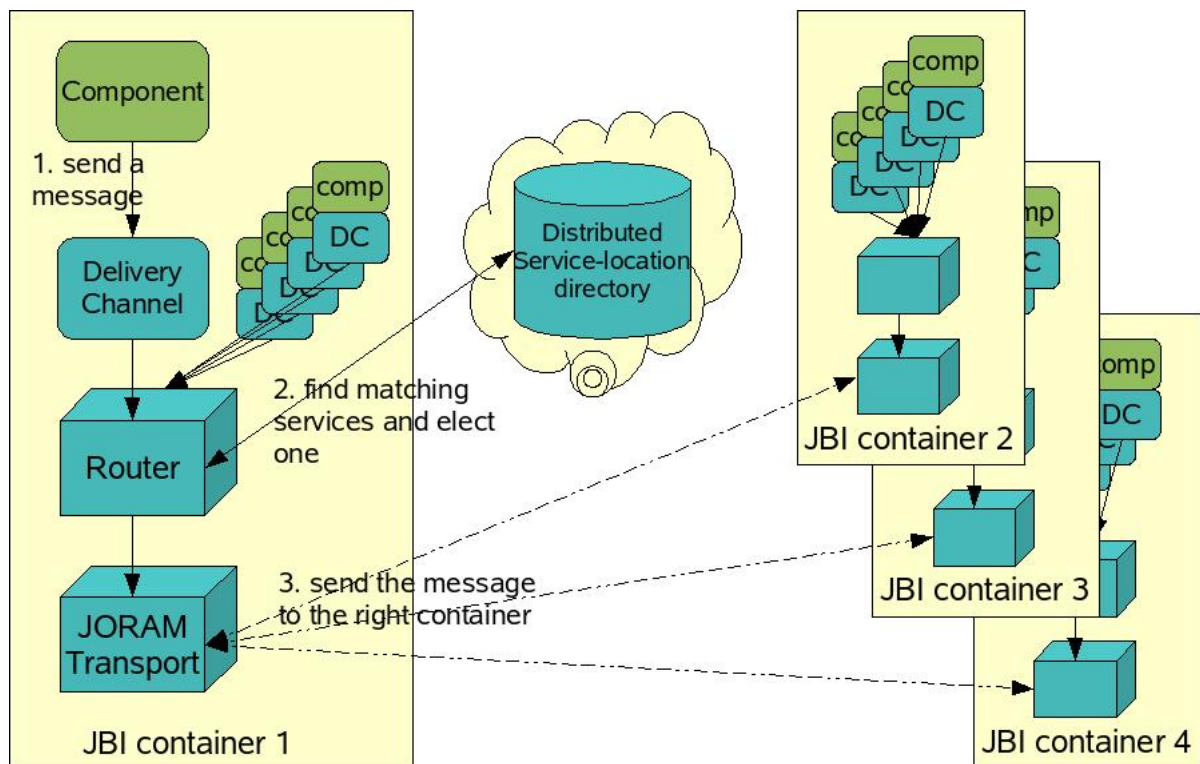


Figure 20: Current Messaging Implementation inside Petals

3.3.2 Administration

The Administration FRACTAL components (Installation, Deployment and Admin services) strongly use the Lifecycle FRACTAL component to manage :

- installation and uninstallation of the JBI components
- deployment and undeployment of ServiceAssemblies

As the Installation service is in charge of creating all the environment for a JBI component, it also uses ClassLoader, Repository and other FRACTAL components.

Moreover, the Autoloader FRACTAL component uses these FRACTAL components to perform the hot (un)deployment of JBI components and service assemblies. In the same way, the SystemState FRACTAL component uses these FRACTAL components to redeploy and restart JBI components or serviceAssemblies when the PETALS container restarts after a stop or a crash.

Finally, the PetalsAdmin FRACTAL component uses those components to perform extra administration. It uses also the RouterMonitor FRACTAL component to activate monitoring and retrieve information.

4 The JOnES distributed message router

This section is focused on the Normalized Message Router and its extension in the JOnES project. The router is the central element in JBI. Indeed each message exchange go through this component. This section is divided in three parts. The first one describes targeted components. The following part emphasizes the JOnES contribution to this distributed approach. The presentation of DREAM and JORAM, open-source middlewares used, concludes this section.

4.1 Introduction

4.1.1 Targeted FRACTAL components

The NMR, described in JBI specification (see 2.3) , is implemented in a distributed way (see 3.3.1). To be more precise the FRACTAL diagram (figure 21) represents , inside the local PETALS architecture, current interactions between the Router (NMR) and the JORAM Transporter :

1. Messages which must be sent to a distant PETALS environment are transmitted to the JORAM Transporter by the Router. Before sending, the router get the destination of the message by using the endpoint service which scans the JNDI distributed directory.
2. Messages received by the Transporter are simply transmitted to the Router to find the matching JBI component in the local environment.

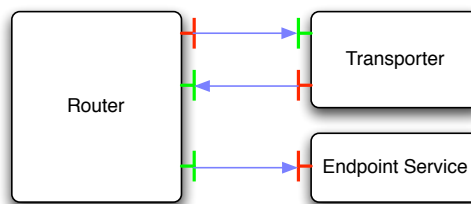


Figure 21: Fractal representation of interactions between Router and Transporter

4.1.2 Final architecture

We extended this first implementation to build a router taking into account concerns related to distributed implementation : security, reliability, performance... These questions are not addressed by the JBI specification as there are no distant communications in the environment. The distribution realized with JORAM cannot resolve all these issues. The following diagram illustrates the added value of the JOnES project : to offer different ways of communication between distant PETALS.

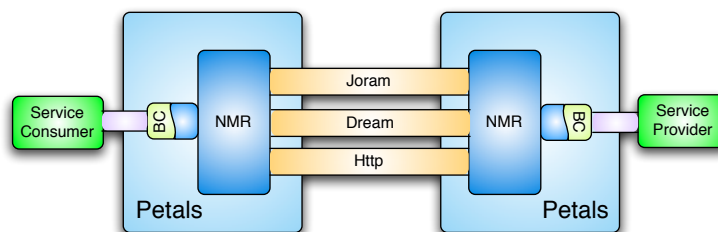


Figure 22: JOnES extension related to NMR

4.2 Towards an Extended Router

This subsection describes the incremental evolution of the router to take into account considerations detailed before. To perform this work, we use DREAM, a communication framework detailed in section 4.3.2.

4.2.1 The Dream Transporter

This section aims at describing the first step of the extension : the construction of a DREAM transporter offering connection-oriented communications. This transporter targets performances of communications as it connects distant environment via TCP sockets.

As shown in the picture below, the DREAM implementation of the transporter transforms the primitive PETALS transporter into a composite components.

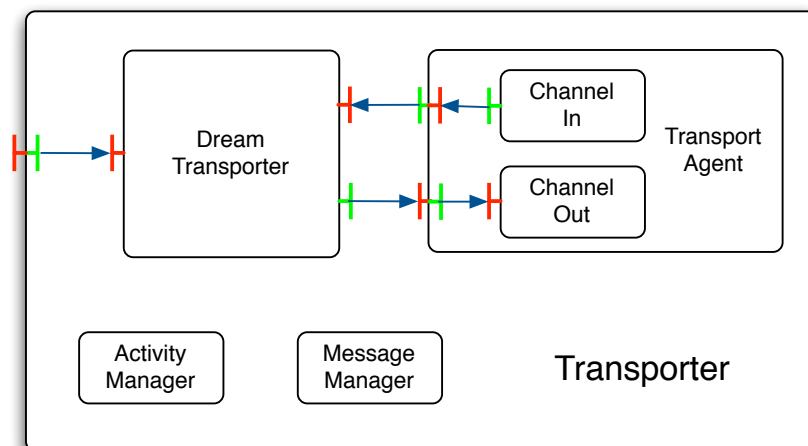


Figure 23: DREAM transporter

The **DREAM Transporter** component implements the `Transporter` PETALS interface and the `Push DREAM` interface. Thus it represents a bridge between the JBI environment (via the Router) and DREAM communications.

The **Transport Agent** component implements the `Push DREAM` interface to send message to a distant PETALS environment via the TCP Protocol. This component is provided by the DREAM framework and manage a pool of TCP connexions.

The **DREAM Managers** are utility components which deal with DREAM messages and activities of components.

4.2.2 Distant Communications with multiple transporters

Once the DREAM transporter was built, the behavior of the Router has been modified. As shown on the figure 24, the router chooses a transporter to send a message to another PETALS environment, JORAM representing the default choice. In addition, the receipt of messages stays unchanged.

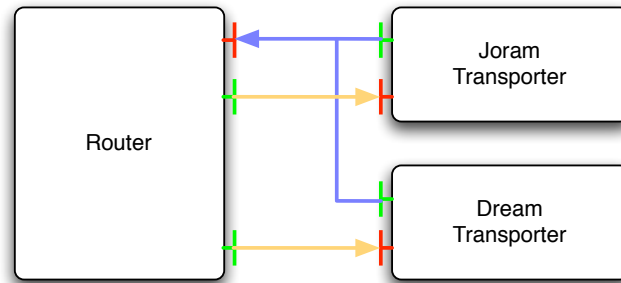


Figure 24: Router with multiple transporters

The pragmatic first approach consists in tagging messages. In consequence, the JBI component which initiates the message exchange has to tag messages if it wants to use a specific transporter. The router just dispatchs messages between transporters.

This simplistic approach, which slightly modifies the router, has two significant disadvantages :

- It makes the distribution non-transparent to JBI components. Indeed a component requiring performance of communications and no reliability has to tag every messages to use the right transporter. This constraint should be avoided for the developer of the JBI component.
- All transporters are launched at startup. It could lead to use useless transporters, especially if the PETALS user knows that PETALS will be launch in a specific context with only one transporter used.

A similar, but less intrusive approach is to map from the MEPs (Message Exchange Pattern) used in the MessageExchange to a specific transporter. For example, InOut could be mapped to Dream in most context, whereas RobustInOnly could be always mapped to JORAM in order to get advantage of JORAM reliability mechanisms.

4.2.3 The JBI Router as Transporter factory

In this section, we describe the architecture of the router, which plays the part of transporter factory. The next subsections distinguish the way to describe transporters to generate. We develop first a static way to describe transporters, then a totally dynamic solution. By the end of the JONES project, only the first part has been totally integrated in PETALS.

Transporters creation based on static description In this paragraph, we model the router as a transporter factory where transporters are described statically (let say in a configuration file). This way to see the router is not really different from the previous approach. However, it offers a way to customize transporters contain in the environment. Thus the PETALS user can avoid the presence of useless transporters.

From the PETALS router point of view, the configuration of transporters must be shared between PETALS instances to avoid useless sending between environment. It can be done via the JNDI distributed directory where information can be stored. The figure 25 illustrates this requirement. The DREAM transporter must not be used because it has not been started on the other side and JORAM neither because of the presence of a firewall blocking matching ports.

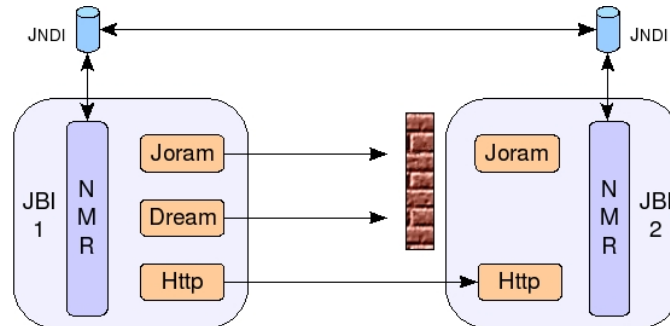


Figure 25: Example of shared configuration

This approach is pragmatic but asks the PETALS user to edit another configuration file. It represents a first step to develop a purely dynamic transporter factory : to manage the creation and initialization of transporters.

Transporters created on the fly Unlike the previous, the approach developed in this section is purely dynamic. The transport can be seen as a service on demand. From this point of view, the initial configuration looks like the current PETALS' one. Only one transporter, implemented with JORAM, is present to realize distant communications : it is the default transporter. If the router received a message on its delivery channel, requiring another transport mode, the matching transporter is created and then bound to the router. The same kind of work is realized by distant environment to create the matching transporter. Once the transporter is ready, the router manages it as in the previous approach.

The diagram 26 illustrates the creation of a dream transporter on the fly.

After receiving a dream-tagged message, the following actions are performed :

1. The local router creates a DREAM transporter
2. It looks for the DREAM transporter in the distant environment (in this example the DREAM transporter is absent)
3. It sends a "create DREAM transporter" via the default transporter
4. The distant router creates a DREAM transporter and sends an ack
5. The local router get the ack and finally sends the message via the DREAM transporter

4.3 Middleware Solutions

As described in previous sections, the distribution inside PETALS rely on two middlewares : JORAM and DREAM. This section describes briefly these tools.

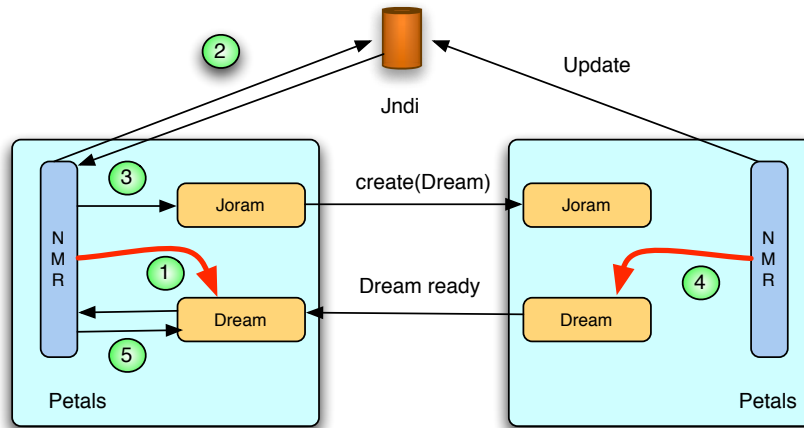


Figure 26: Use case of creation on the fly

4.3.1 JORAM

JORAM [2] is an open-source implementation of the JMS (Java Messaging Service) specification. JMS describes an API that allows Java programs to communicate through asynchronous communication system – i.e. sending and receiving messages. A JMS application consists of three main elements:

- A **JMS platform** (also called JMS provider) that implements the JMS API and provides a set of control and administrative functions.
- The **JMS clients** are application programs, written in Java, that produce and consume messages following the messaging protocol defined in the JMS API.
- The **JMS messages** are entities that allow information to be conveyed between JMS clients. Different types of messages are supported in JMS: structured text (e.g. XML file), binary data, serialized Java objects...

It supports two messaging models:

- **Point-to-Point Model**, based on message queues. A producer client sends a message to a message queue where it is stored temporarily. A consumer client may then read the message from the queue to operate on it. A given message is read only by a single consumer client. The message stays in the Queue until it is read or the message time-to-live expires. The message consumption may be synchronous (explicit call by the consumer client). Message consumption is then acknowledged either by the system or by the client application.
- **Publish/Subscribe model**, based on message Topics. A producer client publishes a message related to a pre-defined Topic. All clients that have subscribed to this topic are notified of the corresponding message.

According to this overview, JORAM is a JMS platform. As any other JMS platform JORAM is structured in two parts: the JORAM server that manages the JMS abstractions (Queues, Topics, ConnectionFactory...) and the JORAM client that is bound to the JMS client application. The JORAM server can be implemented as a centralized service or a set of cooperating services.

4.3.2 DREAM

DREAM [1] 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...

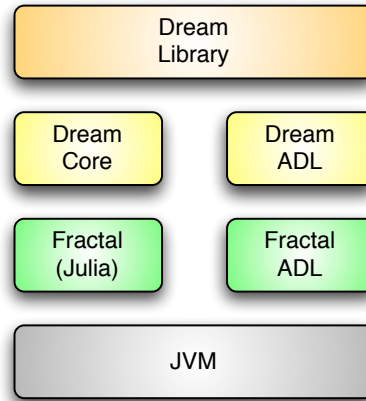


Figure 27: Dream Modules Dependencies

The DREAM project is made of several sub projects:

- The **DREAMCORE** project aims at defining abstractions commonly used in communication middleware (e.g. messages, activities).
- The **DREAMADL** project defines extensions to Fractal's ADL.
- The **DREAMLIB** project provides a library of components that can be used to build communication middleware (e.g. queues, channels, protocols).

Dream rely on Fractal, a component model which provides support for hierarchical and dynamic composition. The main dependencies are depicted in the figure 27.

The DREAM transporter, an example of component build with this framework, is depicted in the figure 23.

References

- [1] Dream website. <http://dream.ow2.org/>.
- [2] Joram website. <http://joram.ow2.org/>.
- [3] Petals website. <http://petals.ow2.org/>.
- [4] Java Community Process. Java(tm) business integration (jbi) 1.0 - jsr 208. Technical report, Sun Microsystems, Inc, August 2005.