Petals Service Platform

Architecture overview

Draft

Adrien LOUIS
08 / 03 / 2005

Summary

Illustrations

1 Macroscopic view

As Petals respects the JBI specification, it can be represented in a macroscopic overview with the following diagram:

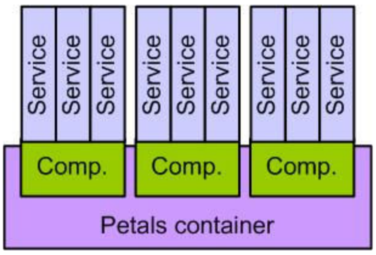

Figure 1 : Petals as a JBI container

A Petals container hosts JBI components, which expose some services.
Description of those services has to be provided by the hosting component in a WSDL2.0 way.
The services are exposed to the JBI environment (i.e. the other components of the Petals environment) through endpoints.

A Petals container interacts with other Petals container in a transparent way for a component. Thus, it allows a component to request a service hosted by a distant component.

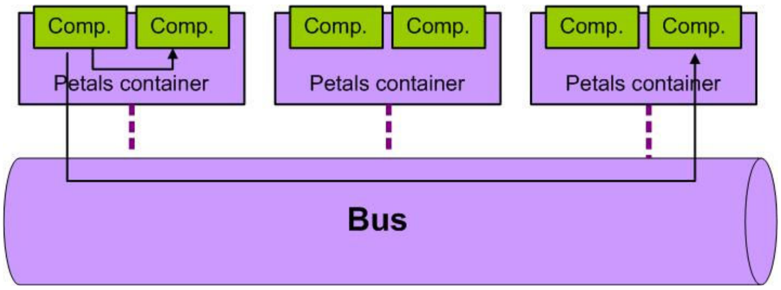

Figure 2 : A ditributed environment

2 Petals core elements

A Petals container is build upon multiple elements:

- The **Petals core** element, which represents the master piece of the Petals container,
- System services components, which are plugged into the container:
    - The **Directory** element, which is the access point of the distributed global directory. It allows the container to register / unregister and to query information about the distributed environment (other containers, services, endpoints).
    - The **ItineraryResolver**, which is in charge of the build of a message itinerary (based on rules), by adding for example a data transformation service between the sender and the receiver of a given message,
    - The **AddressResolver**, which is in charge of finding physical addresses of the services constituting a message itinerary (based on rules), with the help of the Directory component.
    - The **Transporter** element, which is used by the container to send and receive messages to/from other containers,

The **Directory** and the **Transporter** components are directly associated to the container. The **ItineraryResolver** and the **AddressResolver** are associated to a **Router** element, which is associated to the container.

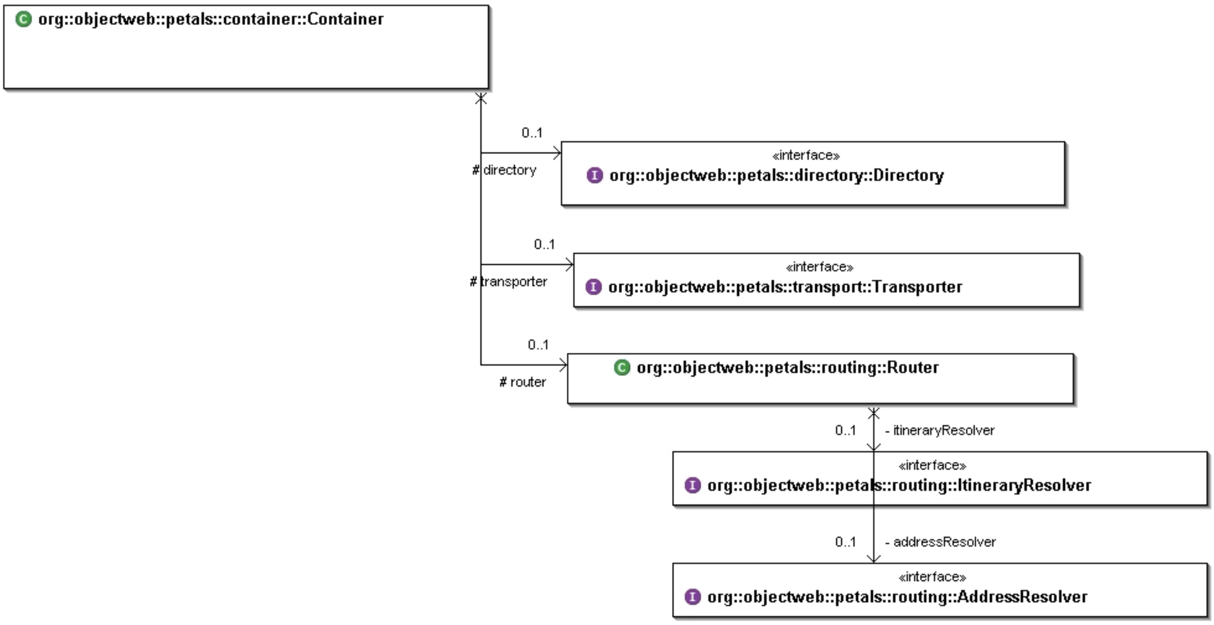
org::objectweb::petals::container::Container
0..1
# directory
«interface»
org::objectweb::petals::directory::Directory
0..1
# transporter
«interface»
org::objectweb::petals::transport::Transporter
0..1
# router
org::objectweb::petals::routing::Router
0..1
- itineraryResolver
«interface»
org::objectweb::petals::routing::ItineraryResolver
0..1
- addressResolver
«interface»
org::objectweb::petals::routing::AddressResolver

Figure 3 : The container and its system services

3 Petals container in action

Start the container

The Petals container hosts the user services, trought JBI components.

JBI components are third part elements, which have to be referenced in the container configuration, and are loaded in the container environment at the initialization of it. They have to respect the *javax.jbi.component.Component* interface to allow the container the possibility to manage them and communicate with them.

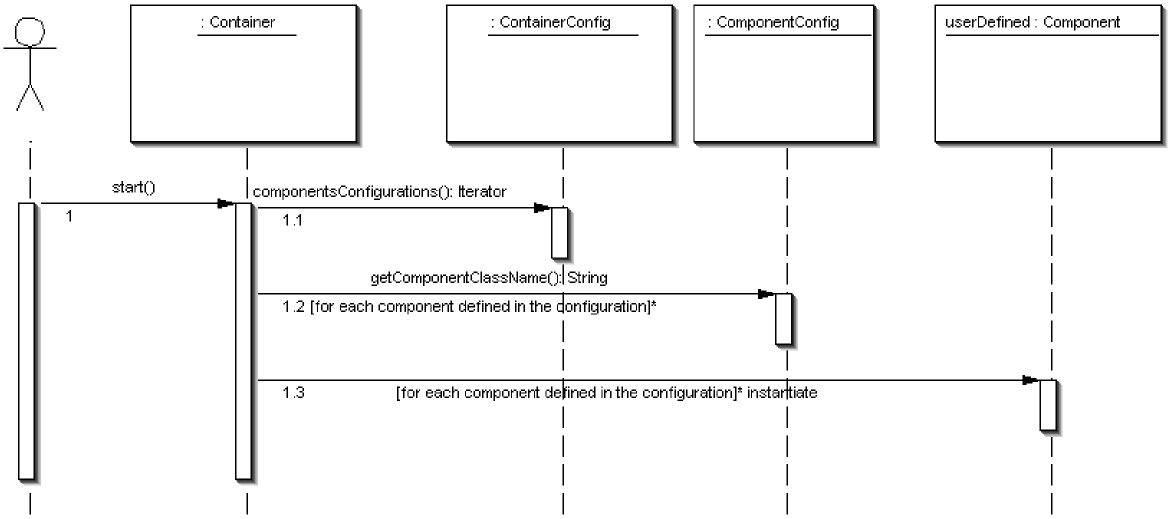
: Container
: ContainerConfig
: ComponentConfig
userDefined : Component
start()
1
componentsConfigurations(): Iterator
1.1
getComponentClassName(): String
1.2 [for each component defined in the configuration]*
1.3
[for each component defined in the configuration]* instantiate

Figure 4 : Instantiation of the binding/engine components

User components, a.k.a. binding and engine components in the JBI specification, are initialized by the container before they start. The container initializes them with a *componentContext* object. This is the element with which components will interact.

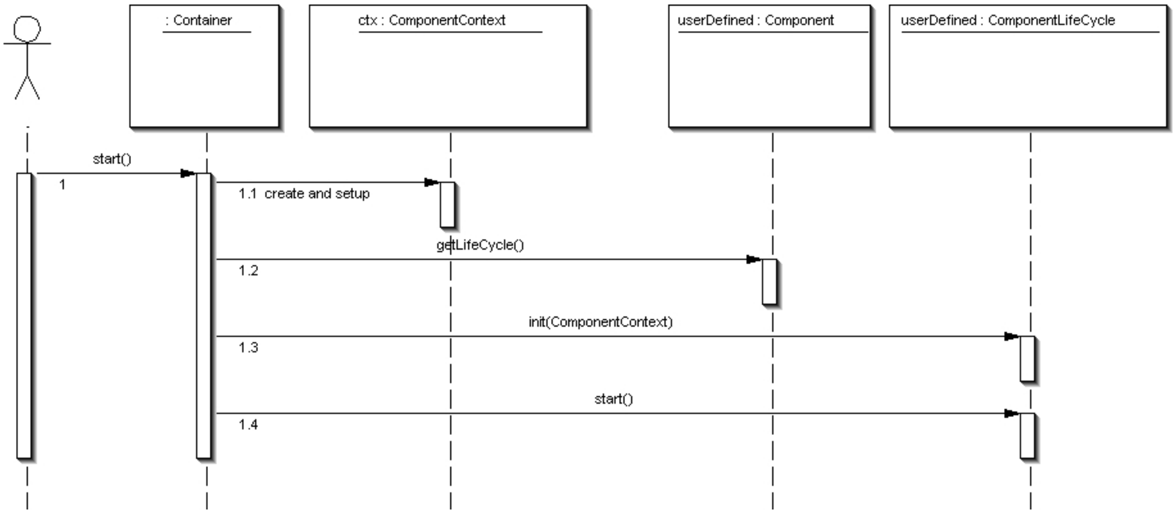

Figure 5 : Start of the components

Register the services

Components have to register in the JBI environment the services they expose, through their *componentContext*. The container uses the **Directory** element to reference them in the distributed directory.

The **Directory** manipulates *Address* elements, which contain all information about the location of a service (the JBI *ServiceEndpoint* description, the container hosting this service and the component in which this *ServiceEndpoint* is based).

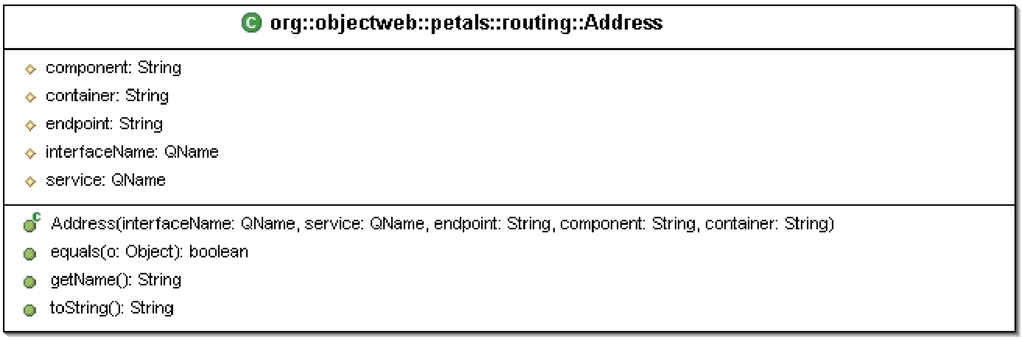
org::objectweb::petals::routing::Address
component: String
container: String
endpoint: String
interfaceName: QName
service: QName
Address(interfaceName: QName, service: QName, endpoint: String, component: String, container: String)
equals(o: Object): boolean
getName(): String
toString(): String

Figure 6 : The "Address" element

Then, those services are ready to be called by other JBI components which are present in the environment (Note that it is the charge of the **Directory** element to promote the registered services in the distributed environment).

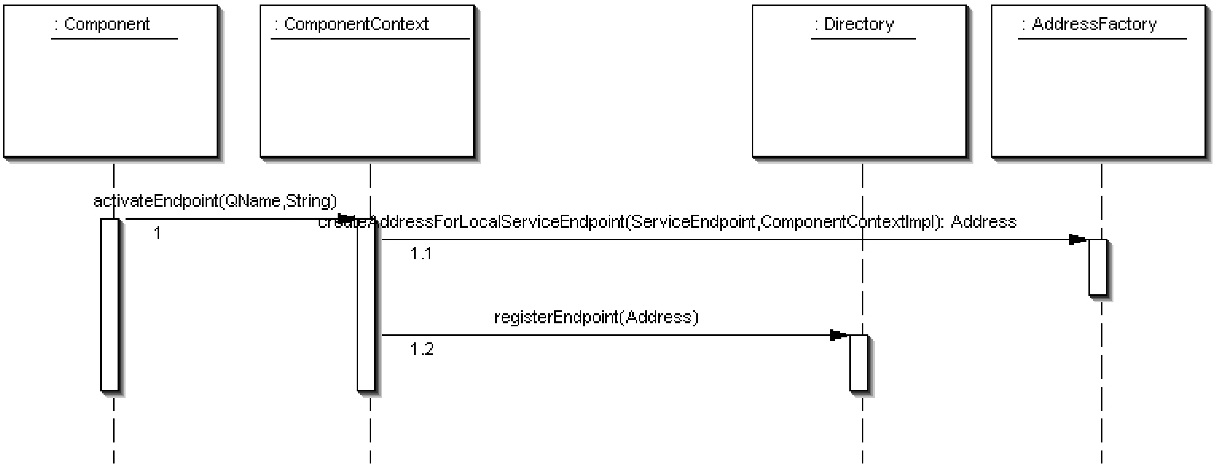
: Component
: ComponentContext
: Directory
: AddressFactory
activateEndpoint(QName,String)
1
createAddressForLocalServiceEndpoint(ServiceEndpoint,ComponentContextImpl): Address
1.1
registerEndpoint(Address)
1.2

Figure 7 : Register a service

Send a message

For a component, the mean of sending a message is to use the *DeliveryChannel* object, which is promoted by the *ComponentContext* object.
The *DeliveryChannel* allow the creation of *MessageExchange* through a *MessageExchangeFactory*. It offer the possibility to create predefined MEP, such as *InOnly*, *InOutput*, *InOptionalOut* and *RobustInOnly* messages.
Then the component specifies in this message the service that it is requesting, and the message content to send.

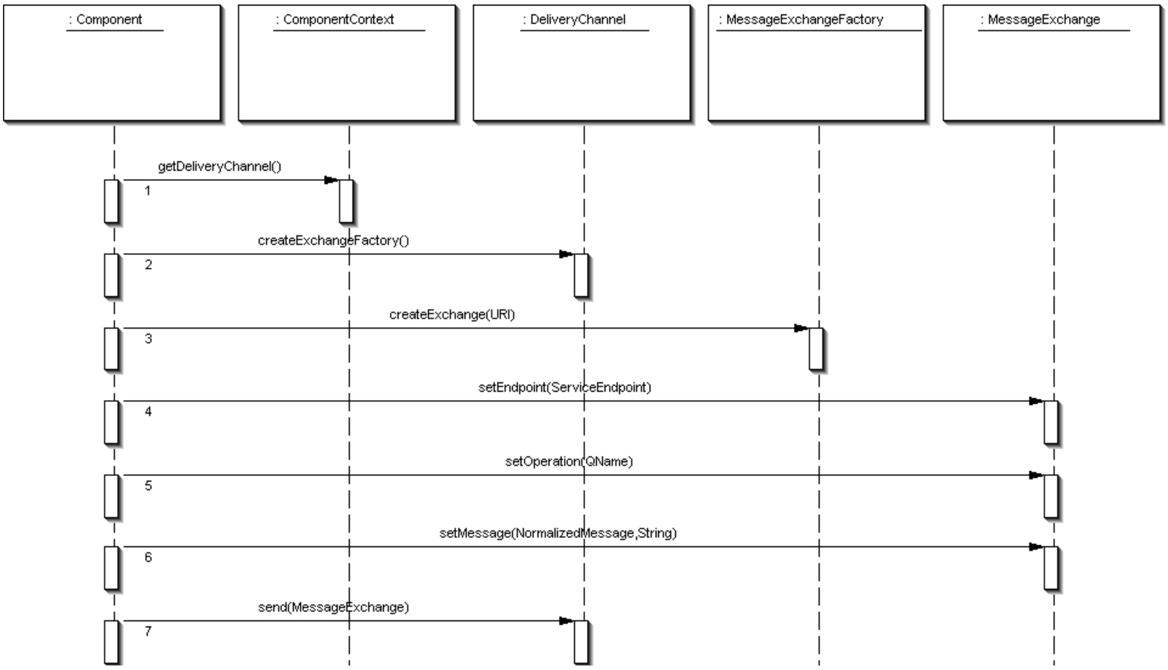

Figure 8 : Send a message – external (component) view

The *MessageExchange* creation is made by the *MessageExchangeFactory* component.
An empty itinerary is then created and attached to the *MessageExchange*. Only the sender address is specified in the itinerary.

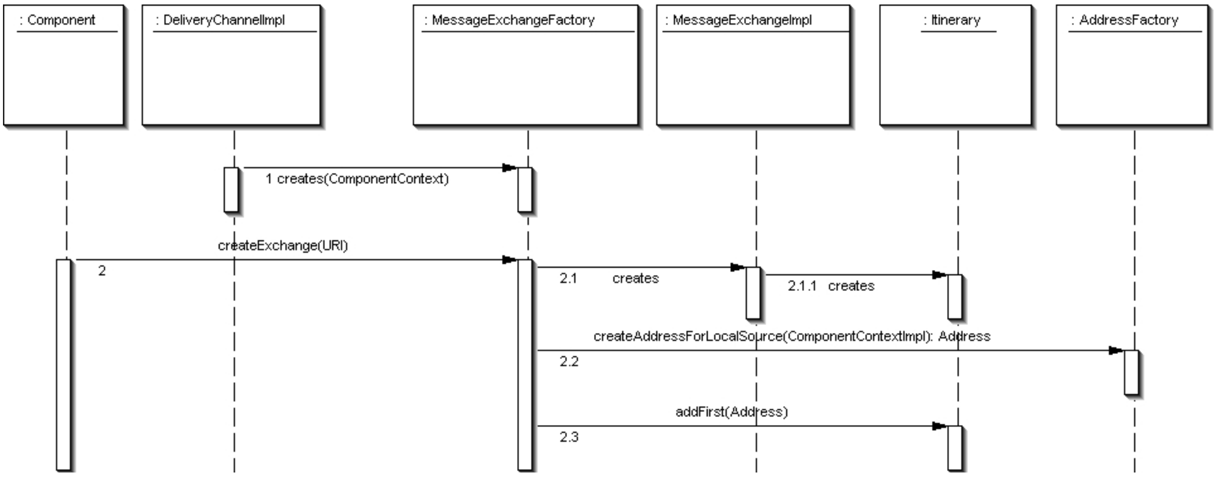

Figure 9 : Create a message and its itinerary

When the component asks the *DeliveryChannel* to send its message, the *DeliveryChannel* send it to the **Router** element.
First, the **Router** asks the **ItineraryResolver** to manage the itinerary of the message.
The **ItineraryResolver** uses rules to determine the itinerary of the given message.

Some examples of itinerary management can be:

- if the message has no recipient, send the message to a Content Based Routing service,
- if the message is written in French and the recipient is English, send the message to a translation service before send it to the final recipient,
- …

Then, the destination of the message (the next node of the itinerary) has to be found.
The **Router** asks the **AddressResolver** to resolve the address of the destination, which consist in finding a physic endpoint satisfying the destination address.

The **AddressResolver** has to find the container, the component and the endpoint name of the destination.
If multiple real destinations satisfy the destination (for example, a translation service is hosted by three containers), the **AddressResolver** uses rules to defined which real destination has to be chosen.

Last, the **Router** asks the **Transporter** to send the message to the chosen real address.

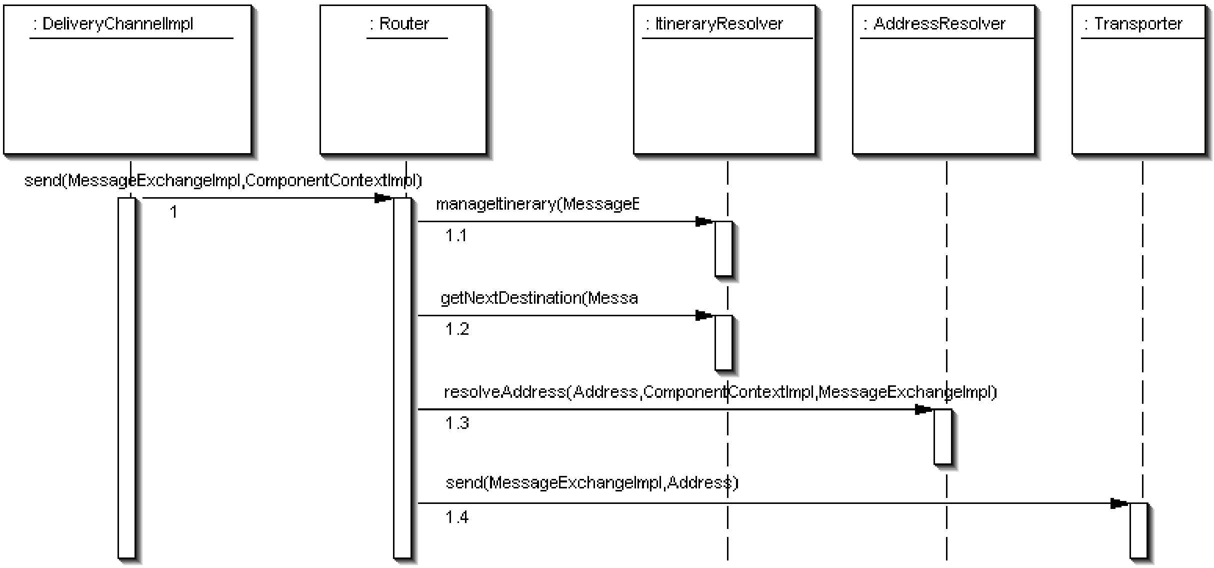
: DeliveryChannelImpl
: Router
: ItineraryResolver
: AddressResolver
: Transporter
send(MessageExchangeImpl,ComponentContextImpl)
1
manageItinerary(MessageE
1.1
getNextDestination(Messa
1.2
resolveAddress(Address,ComponentContextImpl,MessageExchangeImpl)
1.3
send(MessageExchangeImpl,Address)
1.4

Figure 10 : Send a message - internal view

Receive a message

When the **Transporter** of a container receives a message, it calls the **IncomingMessageDispatcher** and gives to it the received message.

The **IncomingMessageDispatcher** look at the message and find the component that has to receive the message.
Then, it pushes the message in the component's incoming messages queue.

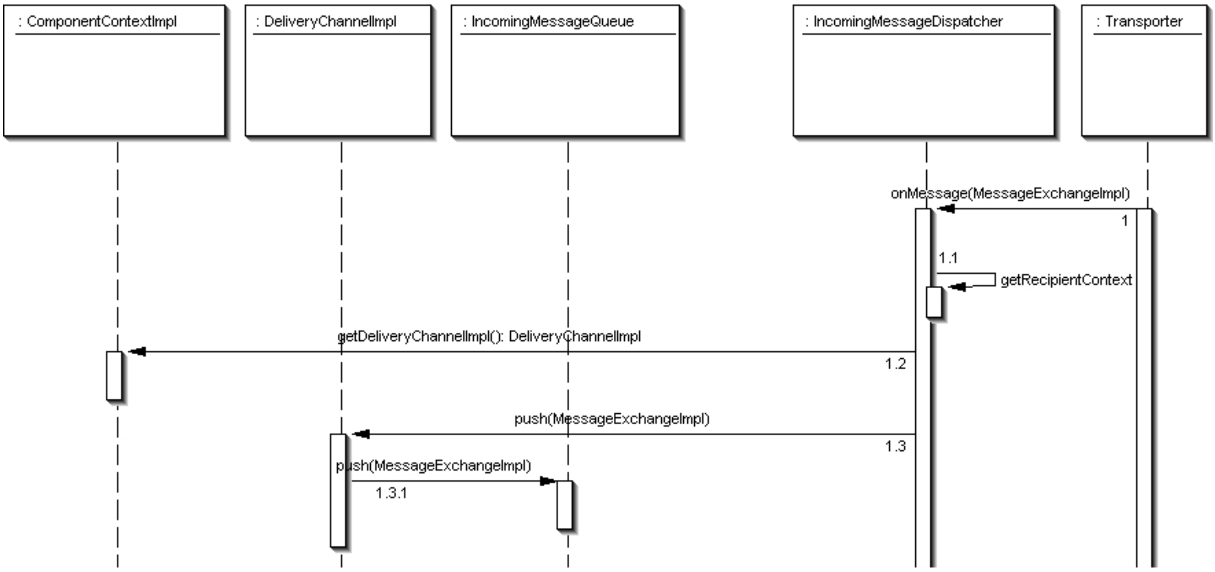
: ComponentContextImpl
: DeliveryChannelImpl
: IncomingMessageQueue
: IncomingMessageDispatcher
: Transporter
onMessage(MessageExchangeImpl)
1
1.1
getRecipientContext
getDeliveryChannelImpl(): DeliveryChannelImpl
1.2
push(MessageExchangeImpl)
1.3
push(MessageExchangeImpl)
1.3.1

Figure 11 : Receive a message - internal view

When the component wants to read received messages, it calls the **DeliveryChannel**. The **DeliveryChannel** looks at its incoming message queue and return the oldest message of the queue, if any.

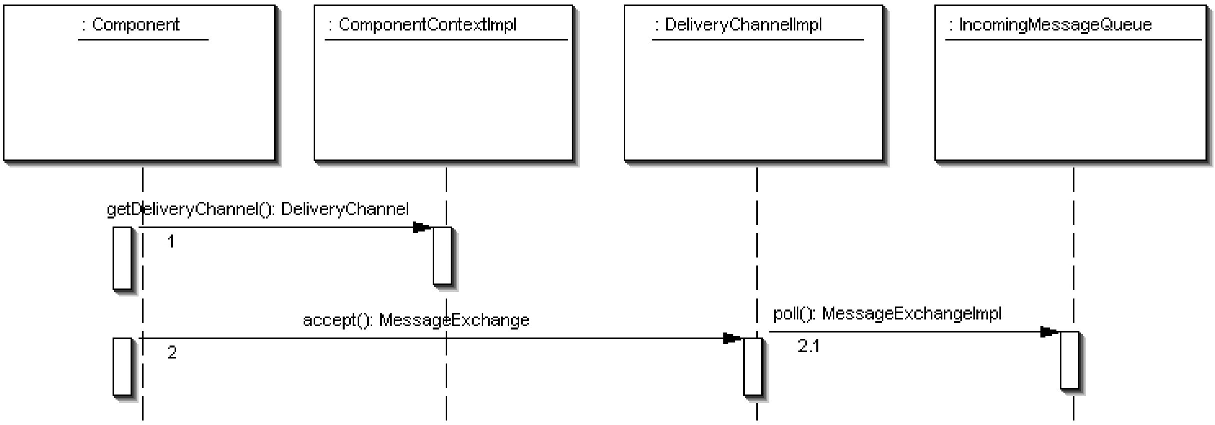
: Component
: ComponentContextImpl
: DeliveryChannelImpl
: IncomingMessageQueue
getDeliveryChannel(): DeliveryChannel
1
accept(): MessageExchange
2
poll(): MessageExchangeImpl
2.1

Figure 12 : Receive a message - external (component) view