JBI Components: Part 1 (Theory)

Ron Ten-Hove, Sun Microsystems

Copyright © 2006, Sun Microsystems, Inc.

1 Introduction

JBI components are where the SOA rubber hits the road: they provide and use the services that are at the heart of service-oriented architecture. Components must bridge the gap between a services model based on message exchange and the particular technologies that the component uses to define or utilize functions.

Before using or creating JBI components, it is very useful to understand the basics of their operation. This article outlines the "theory of operation" for JBI components. A follow-up article, *JBI Components: Part 2 (Practice)*, details how such components can be created in practice.

2 Theory of Operation

Before discussing the details of how JBI components operate, it is useful to put components into their proper context. JBI components are part of a JBI run-time environment, which is discussed in the next section. A JBI environment, populated with JBI components (plug-ins) creates a service-oriented architecture (SOA). SOA is discussed in the section 2.2, after a short discussion of JBI.

2.1 Java Business Integration

JBI is a Java standard for structuring business integration systems along SOA lines. It defines an environment for plug-in components that interact using a services model based directly on WSDL 2.0.

The basic messaging architecture of JBI is illustrated in figure 1. The light blue rectangles represent plug-in components, within the JBI environment (the light grey rectangle). The plug-in components are linked to a message router by delivery channels (the yellow rectangles).

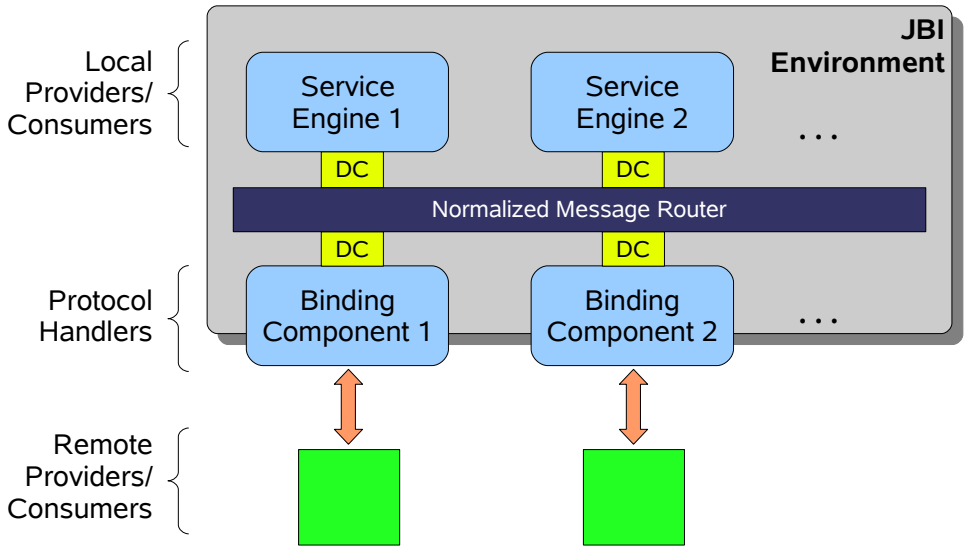

Figure 1: JBI Messaging Architecture

The plug-in components function as service providers, or service consumers, or both. Components that supply or consume services locally (within the JBI environment) are termed "service engines." Components that provide or consume services via some sort of communications protocol or other remoting technology are called "binding components." This distinction is important for various pragmatic reasons, but is secondary to the components' roles as providers and consumers of services.

JBI components, once installed in the JBI environment, interact with each other via a process of message

exchange, which is fully described using Web Services Description Language (WSDL) documents, published by the service provider. These service descriptions are the sole source information needed for service consumer components to interact with the service provider. JBI provides a lightweight messaging infrastructure, known as the normalized message router, to provide the mechanism for actual exchange of messages in a loosely-coupled fashion, always using the JBI implementation as an intermediary.

2.2 Service-Oriented Architecture

SOA is a software system structuring principle based on the idea of self-describing service providers. In this context, a service is a function (usually a business function) that is accomplished by the interchange of messages between two entities: a service provider, and a service consumer.

A service provider publishes a description of the services it makes available. A service consumer discovers and reads the service description, and, using only that description, can properly make use of the service, by mechanism of message exchange. This is illustrated in figure 2. Note that the service provider and consumer share only two things: the service description, and the message exchange infrastructure.

A major advantage of SOA is decoupling: the interface between the consumer and provider is very small. This permits controlled changes to the provider, that will not have unforeseen effects on the consumer. It also allows the consumer to switch providers easily, provided only that a compatible service interface is provided. Decoupling allows changes to occur incrementally.

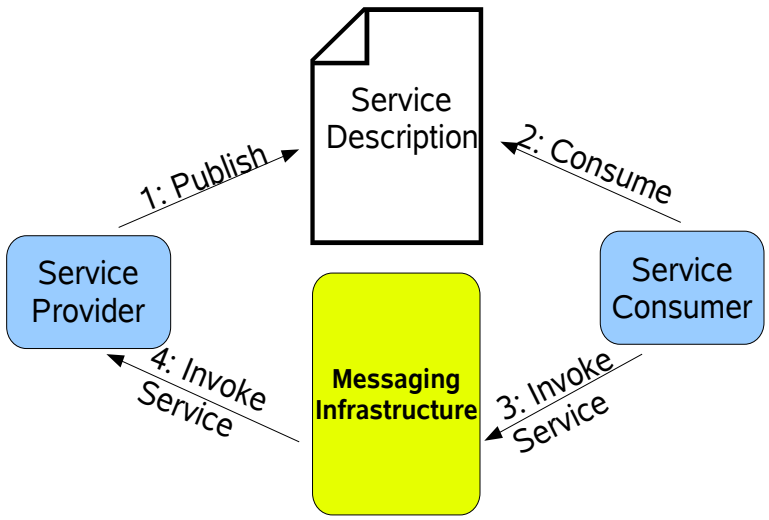

Figure 2: Service Consumer/Provider Interactions

2.2.1 WSDL Model

Service descriptions is JBI are made using the web services description language (WSDL). JBI uses a services model that is based on WSDL 2.0. Descriptions are published by service provider components, and read by service consumer components.

As shown in the figure above, service invocation is accomplished by a service consumer by sending an appropriate request message, via a messaging infrastructure (in this case JBI), to the service provider. Interaction between consumer and provider is always by means of messages. Thus, if the service provider is to provide a response to the service invocation, it will be in the form of a message. If the service provider wishes to return an error (fault) response to the request, it also is in the form of a message.

JBI supports a fixed set of message exchange patterns: a well-defined sequence of message exchanges between the consumer and the provider. A message exchange pattern (or MEP) no matter what contents (or type) of the messages themselves. By supporting a limited set of MEPs, the JBI standard ensures that components have a simple, well known interaction model to implement, ensuring interoperability.

As illustrated in figure 3, individual functions are called "operations" in WSDL and JBI. An operation is defined in terms of message types, and a MEP. Message types are defined in terms of some sort of XML schema language, usually XML Schema 1.0 or RelaxNG. Regardless, messages are XML.

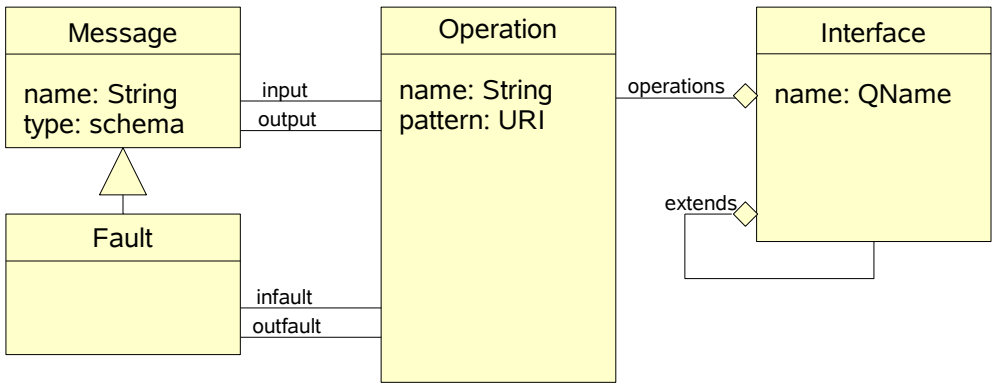

Figure 3: WSDL Abstract Service Model

Related operations are aggregated in an interface. Interfaces may be hierarchically organized, in a manner similar to the the semantics of the Java "extends" keyword.

Figure 3 illustrates the abstract services model of WSDL. This is mapped to the concrete services model, so that clients may access the service via *endpoints*. This model is shown in figure 4. The interface (from the abstract model) is implemented by a service instance. (This is analogous to a Java class implementing a Java interface). The service includes one or more endpoints, which are used by clients to access the service. Examples of endpoint types include SOAP over HTTP, JMS, and REST (XML over HTTP). The concrete model includes type information in the form of a Binding instance that can be shared by multiple endpoints of the same type.

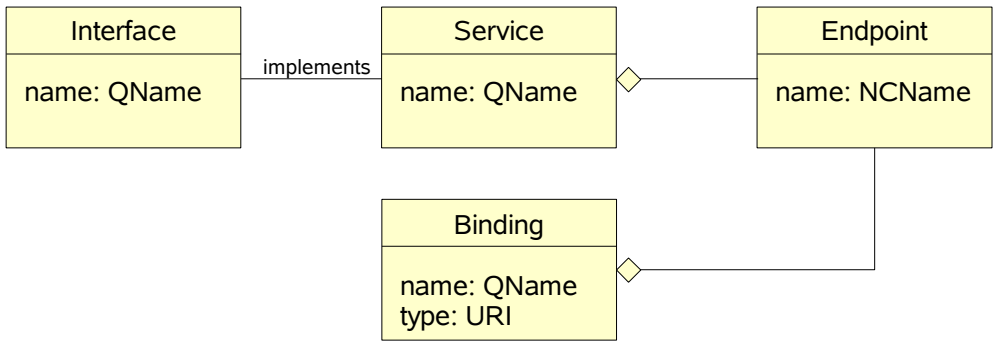

Figure 4: WSDL Concrete Service Model

Endpoints are typically not that important to a JBI consumer component; a desired service and operation are indicated by using the interface name and the operation name. The JBI normalized message router (NMR) serves to find the correct endpoint to send a message to. The NMR is the core of the JBI messaging infrastructure.

2.3 Messaging Infrastructure

JBI provides a messaging infrastructure for components to interact, as illustrated in the SOA interaction graphic shown in figure 2, above. The basic elements of the infrastructure are:

- `NormalizedMessage`. This JBI interface defines the message container that is used to hold service invocation messages, response messages, and fault messages, which are all XML messages. A normalized message includes properties (message metadata), and binary attachments associated with the message.

- `MessageExchange`. This serves as a container for the messages and state of a service invocation (i.e., "calling" a particular operation). There are separate `MessageExchange` subtypes for each supported MEP. New message exchange instances are created using a JBI-provided `MessageExchangeFactory`.
- `DeliveryChannel`. JBI provides each component with a delivery channel, which is the mechanism used to send and receive `MessageExchange` instances.

The JBI implementation provides the logic to route and deliver `MessageExchange` instances that are sent by component through their delivery channels. This is referred to as the Normalized Message Router (NMR).

2.4 Roles of a JBI Component

A JBI component can function as a service provider, or a service consumer, or both. These roles have distinct responsibilities:

- Provider:
    - Activate the service provider endpoint (service name, endpoint name).
    - Provide service description data (WSDL) describing the provided service.
    - Receive message exchanges invoking or continuing an on-going operation. Respond to them according to the MEP and state of the message exchange instance.
    - Perform the function requested in received operation invocations.
- Consumer:
    - Discover the services needed (that is, find the service description). This can occur at design-time (static service dependency), or at run-time (dynamic service dependency).
    - Invoke a needed operation from the service, as dictated by internal processing within the consumer component.
    - Receive message exchanges and respond to them according the MEP and state of the message exchange instance.

2.5 Message Exchange Patterns

As mentioned above, JBI components interact by means of message exchange, using the JBI implementation as the messaging infrastructure. As shown in figure 3, each operation that a service provider makes available has a particular message exchange pattern (MEP) associated with it. An MEP defines the sequence, direction, and cardinality of all message exchanges that occur in the course of invoking and performing the operation. For example, the "in-only" MEP describes a one-way pattern, and is described in detail in the following subsection. JBI defines four MEPs, including in-only. The others are: robust in-only, in-out, and in-optional-out. All four are described in detail in the following subsections.

All message exchanges between consumer and provider are mediated by the NMR. The components interact with the NMR, using their individual `DeliveryChannels`, in two ways:

- Sending `MessageExchange` instances.
- Accepting `MessageExchange` instances.

A consumer component initiates the exchanges by creating a new `MessageExchange` instance, and sending it. Rather than "pushing" the message exchange instance to the provider, JBI defines a "pull" model, where a component accepts `MessageExchange` instances when it is ready.

Once created, an `MessageExchange` instance is sent back and forth between the two participating components,

until the status of the instance is either set to “done” or “error” and sent one last time between the two components. A combination of MEP, state of the `MessageExchange` instance, and role of the component instance tells each component what it is supposed to do in the on-going exchange. This allows components, even ones provided by different component creators, to successfully inter-operate.

This “ping-pong” style of interaction for each pattern type is described in detail in the following subsections.

2.5.1 In-only Message Exchange Pattern

The “in-only” MEP describes a one-way messaging pattern, and is illustrated in the message sequence diagram shown in figure 5.

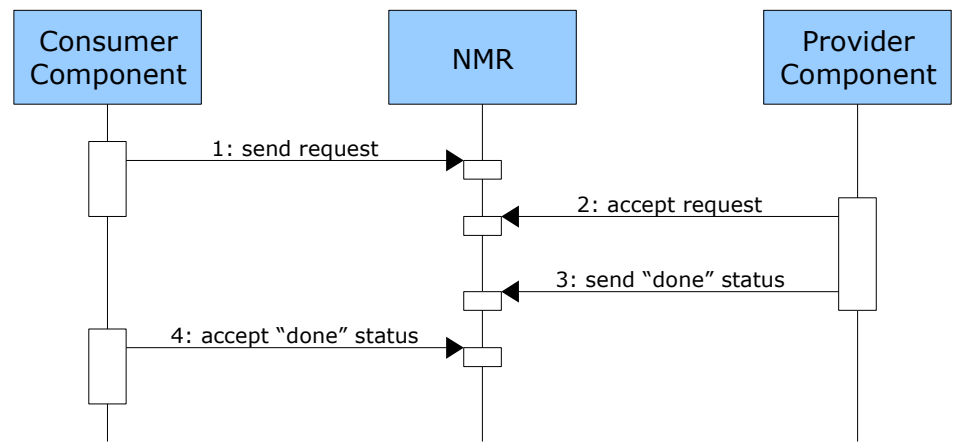

Figure 5: In-only Message Exchange Pattern

The consumer component initiates the pattern by creating a new `MessageExchange` instance, which contains the request (“in”) message, and sending it tow the NMR (step 1).

The NMR determines which provider component is suitable to fulfill the new request, and queues the `MessageExchange` instance for delivery to that provider.

The provider component “accepts” the instance from the NMR (step 2), processes the request message in it, and completes the MEP by sending a “done” status notification to the NMR. (This is accomplished by setting the status of the `MessageExchange` instance to “done”, and sending the instance to the NMR.) The NMR routes the instance back to the request originator (the consumer component), which accepts the notification. This concludes the “life cycle” of the MEP; both components now regard the on-going exchange to be complete.

2.5.2 Robust In-Only Message Exchange Pattern

The robust in-only pattern modifies the in-only MPE by allowing the provider to easily return error responses to in-only message exchanges. If the provider finds no fault with the “in” message from the consumer (the request message), the message sequence is exactly the same as that shown in figure 5. If the provider wishes to return a fault, then the sequence is as shown in figure 6. If the provider wishes to return a fault

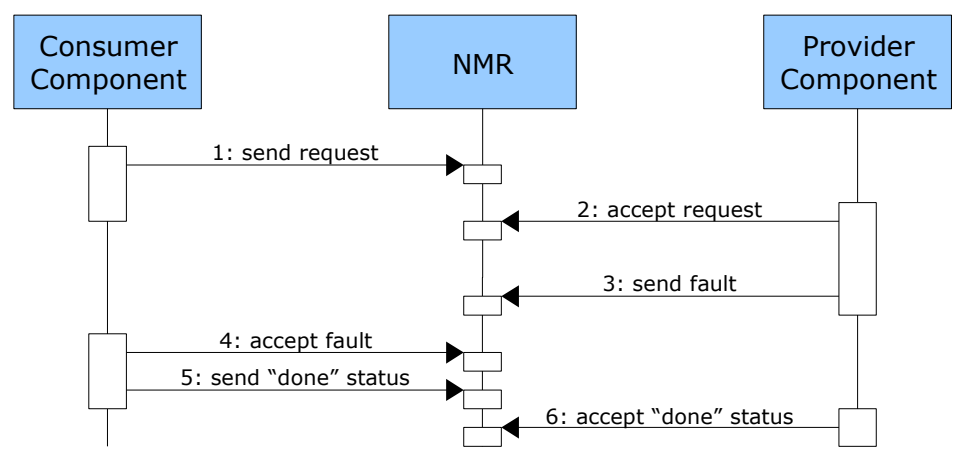

Figure 6: Robust In-only Message Exchange Pattern (fault scenario)

instead, then the sequence is as shown in figure 6. At step 3 the provider sends back a fault message (instead of a "done" status in the normal case). The consumer accepts this fault from the NMR, and closes the message exchange by sending a "done" status to the provider. The provider accepts this from the NMR, completing the message exchange's life cycle.

2.5.3 In-Out Message Exchange Pattern

The in-out MEP is a two-way messaging pattern, where the response from the provider can be either a normal message or a fault message. The "normal" response case is illustrated in figure 7. The consumer component initiates the pattern by creating a new `MessageExchange` instance, setting the "in" (request) message in the instance, and sending it to the NMR (step 1). The provider NMR selects the appropriate provider component, and queues the instance for delivery to that component. The provider accepts the instance, processes the request message, sets the "out" (response) message of the instance, and finally sends it to the NMR (step 3). The NMR queues the instance for delivery to the consumer component. The consumer accepts the instance (step 4), and ends the MEP by setting the instance's status to "done" and sending it to the NMR (step 5). The provider accepts the instance from the NMR (step 6), indicating the end of the MEP and the end of the instance's life cycle.

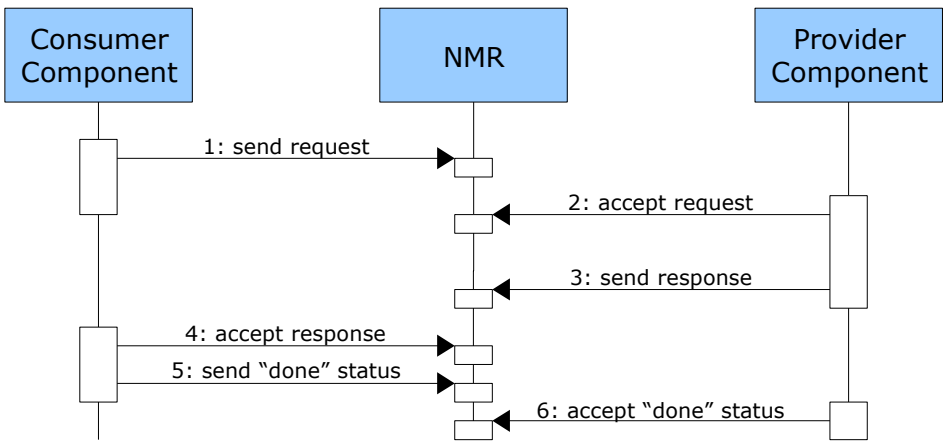

Figure 7: In-Out Message Exchange Pattern (normal response)

Alternatively, the provider component can choose to return a fault message at step 3. This sequence is shown in figure 8. Other than substituting a fault message for the normal response message, the message sequence is the same.

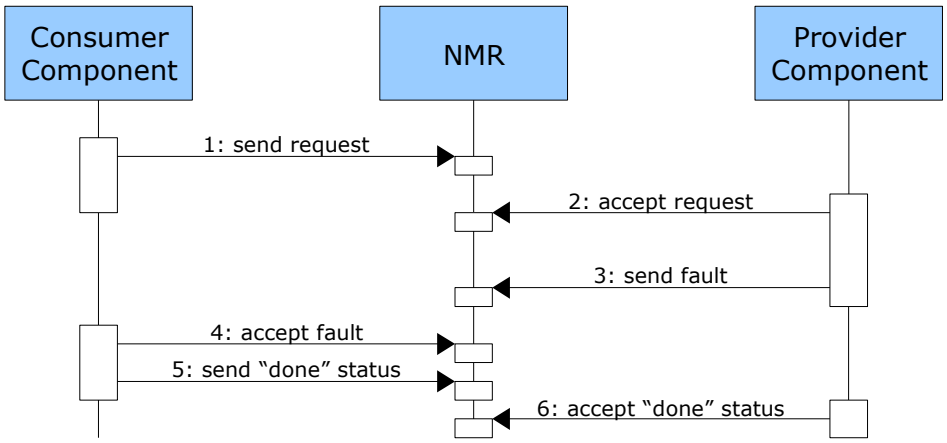

Figure 8: In-Out Message Exchange Pattern (fault response)

2.5.4 In-Optional-Out Message Exchange Pattern

This pattern improves on the in-out MEP by making the message response by the provider optional. This provides greater flexibility than either the in-out or robust in-only patterns, but at the expense of greater complexity. In this pattern, the provider can respond to a new request message in three ways: with a normal message, a fault message, or a "done" status (i.e., the output is optional). If the provider responds with a normal message, the consumer gets a choice: respond with a "done" status, or with a fault message of its own (this pattern allows for an immediate error response from a consumer to a normal response from the provider). This pattern supports four different possible sequences, as illustrated in figures 9 through 12.

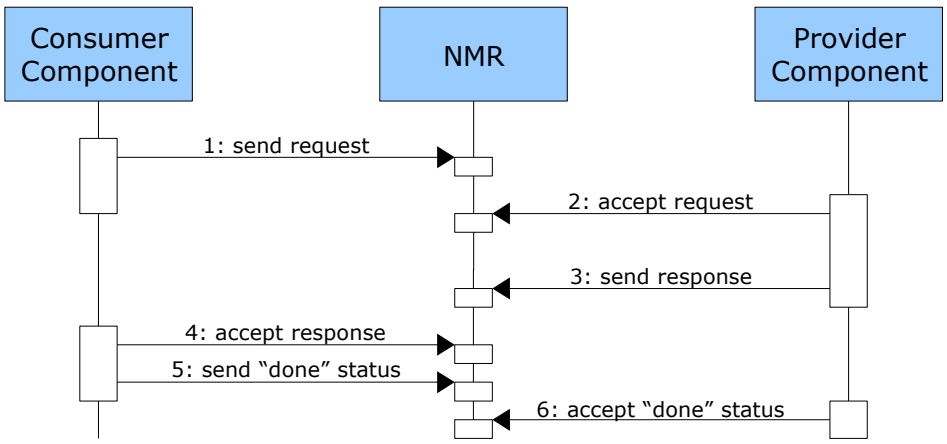

Figure 9: In-Optional-Out Message Exchange Pattern (normal response 1)

The first variant, illustrated in figure 9, involves the provider deciding to send a normal response message, and the consumer deciding to accept it (not to send a fault). (This sequence is identical to the in-out MEP shown in figure 7.)

A variation of the above sequence occurs if the consumer elects to send a fault message in response to the response message. This is illustrated in figure 10. As always, the consumer component initiates the sequence by sending a request message in a new `MessageExchange` instance to the NMR (step 1). The provider accepts the instance (step 2), sets the "out" (response) message of the instance, and sends it back to the NMR (step 3). The consumer accepts the instance (step 4), analyses the response message, and determines that the response is not acceptable (based on application criteria). It creates a fault message, and uses it to set the "fault" message of the `MessageExchange` instance. It sends the instance to the NMR (step 5), which is accepted by the provider (step 6). The provider, after processing the fault message, sets the instance's status to "done", and sends the instance to the NMR (step 7), indicating the end of the exchange. The consumer accepts the exchange (step 8), thus completing the MEP.

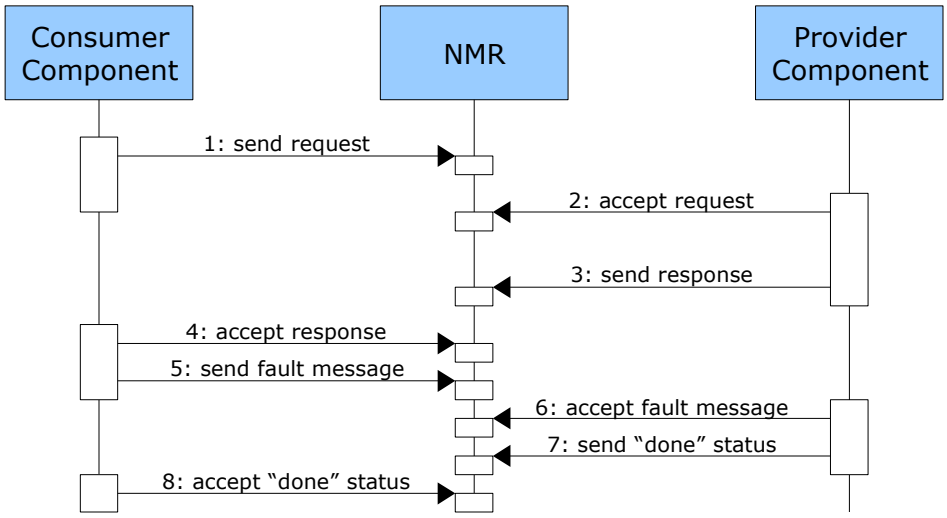

Figure 10: In-Optional-Out Message Exchange Pattern (normal response 2)

The provider can return a fault message in response to the "in" message. This sequence is the same as the pattern *In-Out Message Exchange Pattern (fault response)*, shown in figure 8. The sequence for the In-Optional-Out MEP with a provider fault response is shown in figure 11. The consumer initiates the sequence by creating a new `MessageExchange` instance, with an "in" (request) message, and sending the instance to the NMR (step 1). The NMR selects the appropriate service provider, and queues the instance for delivery. The provider component accepts the instance (step 2), processes it, and elects to response with a fault message. This message is set in the instance, and the instance sent to the NMR (step 3). The consumer accepts the instance from the NMR (step 4), processes the fault response message, sets the "done" status of the instance, and sends it to the NMR (step 5), thus ending the MEP. The provider accepts the instance (step 6), and processes the status change, ending the sequence.

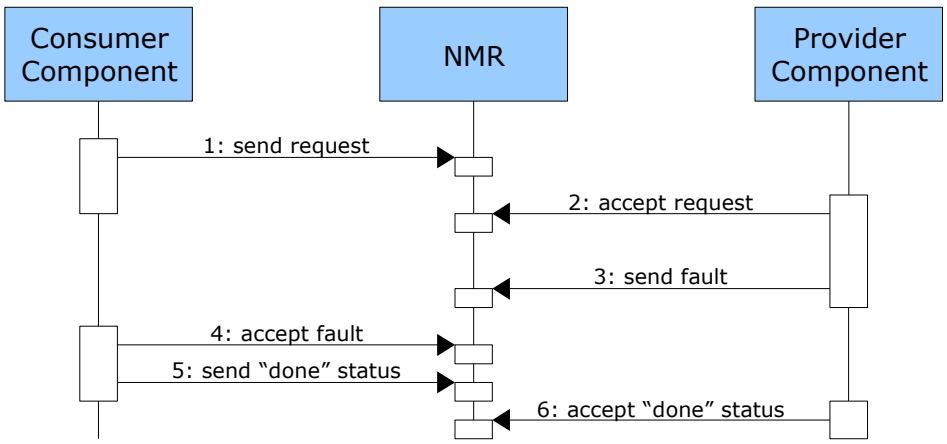

Figure 11: In-Optional-Out Message Exchange Pattern (provider fault response)

The final variation in provider responses is shown in figure 12. This is the same sequence as the In-only MEP, shown in figure 5. This sequence is initiated by the consumer component, which constructs a new `MessageExchange` instance and sets its "in" (request) message. The consumer sends the instance to the NMR (step 1). The NMR determines which provider component should be used, and queues the instance for delivery. The provider component accepts the instance (step 2), processes the request, and elects to return a "done" status. It sets the instance status to "done", and sends the instance to the NMR (step 3), indicating the end of the MEP. The consumer accepts the instance, and processes the status change of the instance, ending the sequence.

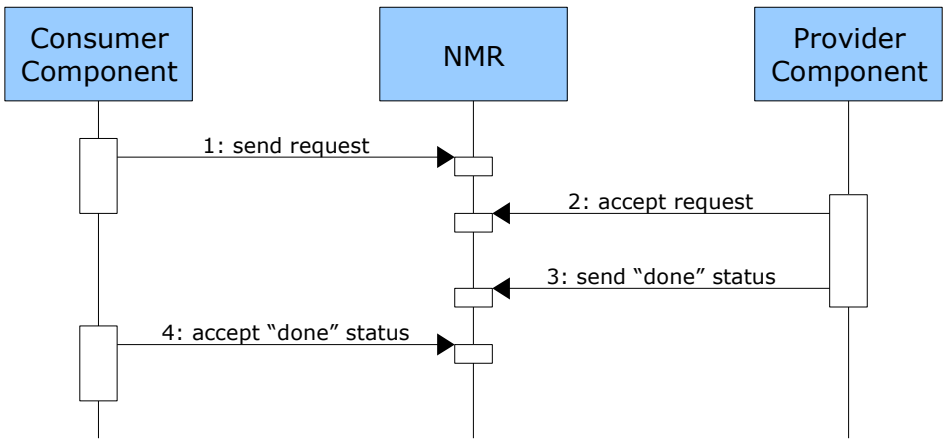

Figure 12: In-Optional-Out Message Exchange Pattern (provider "done" response)

2.5.5 Message Exchange Instance Status

Each message exchange has a complete life cycle, which is reflected in the status of the instance. When first created, an instance is in the `ExchangeStatus.ACTIVE` state. When this status changes, the MEP is considered complete. Normally, `ExchangeStatus.DONE` is the only terminating state of the instance, regardless whether a fault or normal message is involved.

In the (rare) case where a component that "owns" (has just created or accepted[1]) a `MessageExchange` instance runs into a technical problem that cannot be conveyed by normal messaging (normal or fault), it can abruptly end the on-going MEP by setting the status of the instance to `ExchangeStatus.ERROR`. This abrupt closure can occur at any point in the MEP before the `DONE` status is set.

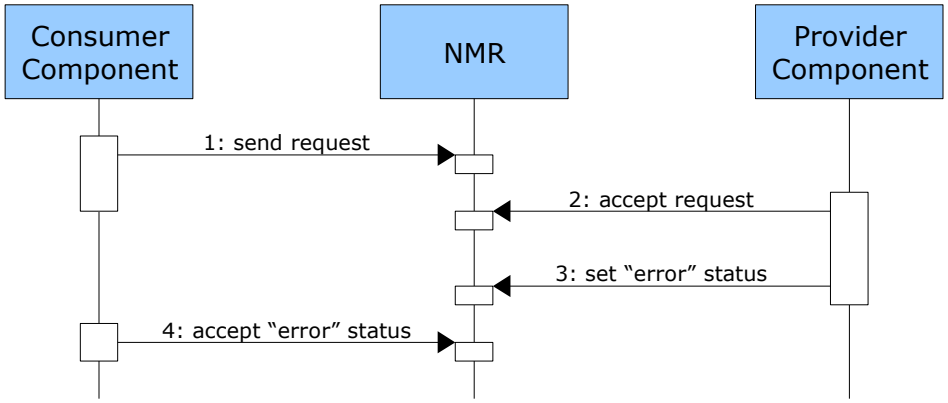

Figure 13: In-Out Message Exchange Pattern (abrupt closure example)

An example of abruptly ending an on-going MEP is shown in figure 13. This shows an in-out MEP. The consumer, as always, initiates the pattern by creating a new instance of `MessageExchange`, with the "in" message, and sending it to the NMR (step 1). The NMR queues the instance for delivery to the provider it selects. The provider accepts the instance (step 2), and, in the midst of processing the request, runs into an unexpected processing error, and cannot proceed further, or compose a suitable fault message. It therefore sets the "error" status of the instance, and sends it to the NMR (step 3), abruptly ending the MEP. The consumer accepts the instance (step 4). Due to its `ERROR` status, the consumer must perform application-level error recovery as appropriate.

The preceding is just an example. All of MEPs can similarly be abruptly ended before the exchange status is set to `DONE`.

2.6 Component Life Cycle

In order to make a system composed of various plug-in components manageable in a consistent fashion, JBI defines a standard life cycle for components. This allows a system administrator to handle a JBI system easily, allowing, for instance, a single command to be used to "stop" all the system, including all components.

The life cycle of a component is depicted in figure 14. A component must supply implementations of two JBI interfaces to allow the JBI implementation to control its life cycle: the `Bootstrap` interface (for `onInstall()` and `onUninstall()`), and the `ComponentLifeCycle` interface, for the `init()`, `start()`, `stop()`, and `shutDown()` methods. These various methods will be called by the JBI implementation such that only the state transitions shown in figure 14 will be made.

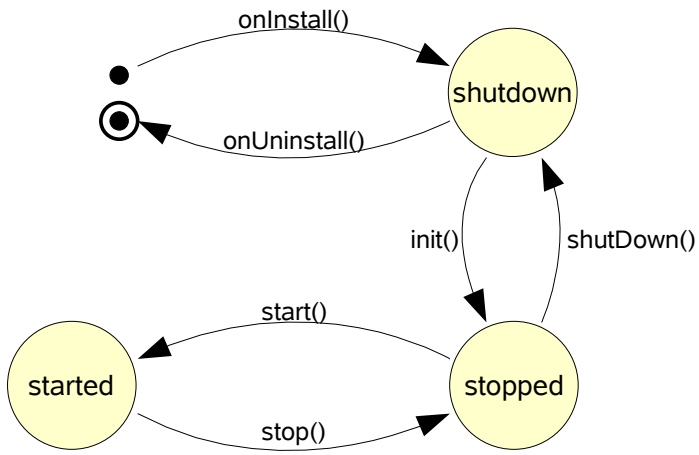

Figure 14: Component Life Cycle (Basic)

A component starts in the so-called empty state: it is either not yet installed, or has been uninstalled. This state is of little interest to the component creator, since the component cannot do anything. After the JBI

1 Only the component that "owns" the `MessageExchange` instance can update its state; when it sends the instance it relinquishes ownership.

implementation finishes installing a component, it creates an instance of the component's `Bootstrap` implementation, and calls its `onInstall()` method.

The “started” state means that the component is actively offering and consuming services (accepting and creating new `MessageExchange` instances via its `DeliveryChannel`).

The “stopped” state means that the component is no longer offering and consuming services. Resources needed to enter the “started” state are allocated (e.g., the `DeliveryChannel` is opened or remains open).

The “shutdown” state means that the component is no longer offering and consuming services, and that all resources used by the component have been released. This state is generally entered prior to full shutdown of the JBI system, or deinstallation of the component.

Generally the transition from “started” to “stopped” should be in an orderly fashion. Ongoing work should be completed during the transition, meaning that message exchanges should still be processed to completion.

2.7 Service Unit Processing

Most interesting components function as containers: artifacts are deployed to them while they are running, adding to their behavior. A simple example is an XML transformation service engine, which uses XSLT as the way of specifying new transformation types that the component offers as a service. In this case, as new XSLT style sheets are deployed to the service engine (plus some extra data describing the new service name), the service can activate new services.

Not all components function as containers. Those that don't either have a fixed set of services offered (and consumed), or use other mechanisms to dynamically vary their behavior (such as using a registry/repository that is managed outside of the component).

When a service unit is deployed to a container component, the component processes the SU in component-specific ways. As a result of the deployment (and start-up) of the SU, the SU provides and/or consumes services. This relationship between a service unit and services is often abbreviated, by referring to the services offered or consumed by the SU.

If a component wishes to serve as a container, it must implement the JBI `ServiceUnitManager` interface. This provides a set of methods that the JBI implementation uses to manage the full life cycle of a deployment. This is depicted in figure 15. This is similar to the component life cycle, but affects only the component behavior related to the service unit itself. The method calls on the transition arcs are all methods from the `ServiceUnitManager`.

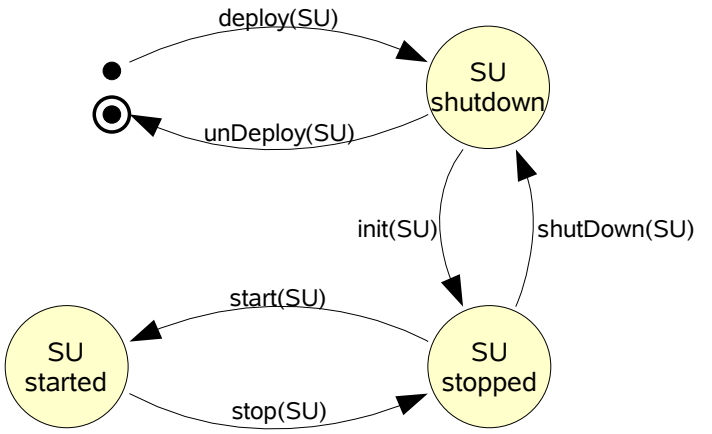

Figure 15: Service Unit Life Cycle

When an SU is in the “Started” state, it provides and consumes services as dictated by the SU contents. When an SU is in the “Stopped” state, those services are no longer provider or consumed. When an SU is in the “shutdown” state it does not consume or provide services, and has released all run-time resources associated with the service unit.

In our transformation service engine example, the service unit consists of an XSLT style sheet. When the SU is deployed to the service engine its `ServiceUnitManager.deploy()` method is called. When the `ServiceUnitManager.init()` method is called, the example engine can precompile the style sheet, getting it ready for use. When `ServiceUnitManager.start()` is called, the transformation service provided by the SU is activated by the

component. When the `ServiceUnitManager.stop()` method is called, the component deactivates the service provided by the SU. When `ServiceUnitManager.shutDown()` is called, the precompiled style sheet and associated resources are released. Finally, when `ServiceUnitManager.unDeploy()` is called, any record of the service unit kept by the component should be erased.

3 Conclusion

Implementing a JBI plug-in component involves fulfilling several JBI-defined contracts. These contracts cover:

- Installation packaging and description
- Component life cycle management
- Messaging based on defined message exchange patterns
- Publication of offered services

and, optionally,

- Service unit management

Foremost, a component maps between its own particular technical domain and the JBI services model. This covers two distinct tasks:

- Providing services to the JBI realm: mapping service requests to appropriate component-domain actions:
    - converting the service-domain request message to component-domain data and actions (this is called denormalization)
    - if necessary, converting the component-domain data generated by the request to a service-domain message (normal or fault; this is called normalization).
- Consuming services from the JBI realm: mapping component-domain actions to JBI-domain service requests:
    - converting service domain actions and data into a service-domain request message (normalization)
    - if necessary, converting the service domain response (normal or fault) to component-domain data.

These two tasks, involving normalization and denormalization of component-domain data, are key to adapting the component's domain to the loosely-coupled, reusable, interoperable domain of standard services.

Implementation of components will be discussed in the second part of this article, entitled *JBI Components: Part 2 (Practice)*.