Using JBI for Service-Oriented Integration (SOI)

Ron Ten-Hove, Sun Microsystems
January 27, 2006

Introduction

How do you use a service-oriented architecture (SOA)? This is an important question for those of us contemplating implementing a SOA, or investigating the potential benefits of using SOA. SOA introduces a new way of structuring applications, by composition of *services* rather than modularized, structured code. Many SOA applications come from the realm of enterprise integration. Indeed, you can regard any SOA application as an instance of service-oriented integration (SOI).

This paper will illustrate some of the principles used in SOI, using Java Business Integration (JBI) as a foundation.

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 1. Note that the service provider and consumer share only two things: the service description, and the message exchange infrastructure.

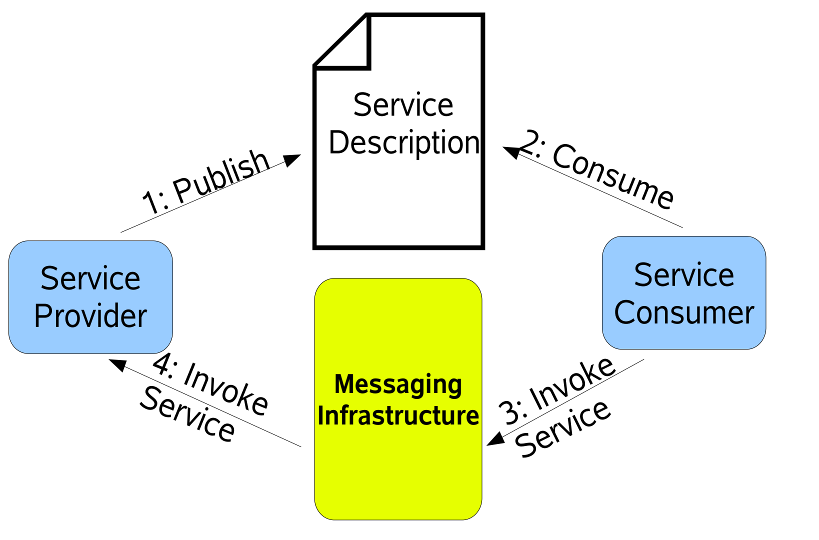

Figure 1: Service Consumer/Provider Interactions

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 that a compatible service interface is provided. Decoupling allows changes to occur.

Another advantage of SOA is reuse: services can be reused in a variety of applications. In fact, given a sufficient palette of available services, one can imagine applications that consist of nothing but services: services used, and services provided. This new application model, termed a “composite application”, leads very naturally to business process automation, where business processes are defined in terms of services provided and consumed. Available services provide the appropriate “steps” needed in business processes. Creation of new business processes in a SOA is quick and easy, as is modification of existing process definitions. Service composition leads very easily to the current “holy grail” of commercial IT: business agility.

Building applications for a SOA environment involves two distinct development activities:

- **Service creation**. This involves integrating existing business IT functions by making them available as services, or creation of new services using existing code-centric development methods.
- **Service composition**. This involves using existing services to create a larger application (or perhaps service) based on a composition of those services. This type of composition is often used to create business processes, and require orchestration of services by means of a business process engine.

Service creation uses skill sets and technologies largely familiar to developers, adding an additional requirement that a service interface be added to the business functions being created. The granularity of services (what functions are exposed, and how coarse-grained they are from a business function perspective) should be dictated by the second form of SOA development: service composition. Service granularity should foster reuse of services needed for composite applications, particularly business processes. Creation of composite applications involves new skills and technologies that most developers do not currently have.

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 2. 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).

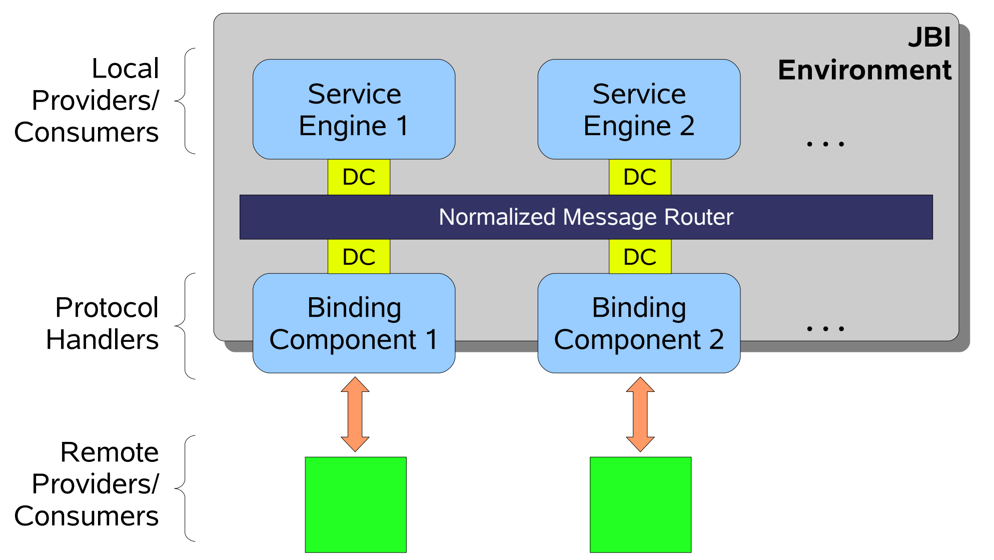

Figure 2: JBI Messaging Architecture

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.

Components as Containers

JBI components are, in the general case, considered to be containers: entities into which other artifacts are "deployed" in order configure them to provide (and/or consume) particular services. For example, an EJB service engine would have EAR files deployed to it in order to add new services implemented as EJBs. A SOAP-over-HTTP binding component would have descriptions of remotely accessible services deployed to it; the component would make those services accessible by consumer components in the JBI environment.

JBI directly supports the deployment of artifacts to plug-in components, using an archive (ZIP file) package called a *service unit*. The contents of a service unit are opaque (to the JBI implementation), except for a single descriptor file named META-INF/jbi.xml. This contains information about the static services that will be provided and consumed once the service unit is successfully deployed into the component. The contents of the service unit are known only to the component that the package is deployed to, and the tools that were used to create the service unit.

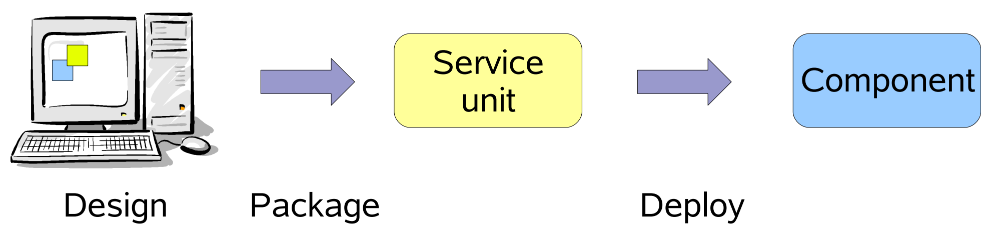

Figure 3: Service Unit Creation and Deployment Work Flow

A service unit can be regarded as a type of "black box", with declared services that it statically requires and provides. This can be depicted graphically, as shown in figure 4. The yellow chevrons on the left-hand side of the service unit box, pointing away from the box, indicate the services provided by the service unit (once deployed). The blue chevrons on the right-hand side of the service unit indicate services that are consumed by the service unit. These chevrons are pointed towards the service unit to indicate a dependency. A service unit can have zero or more of each type of chevron.

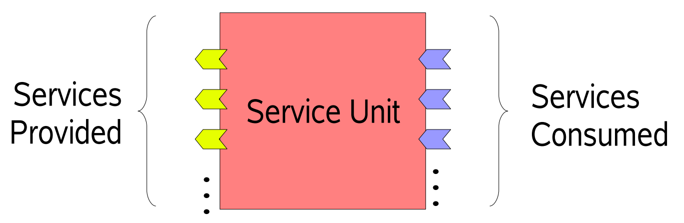

Figure 4: Service Unit: Services Input/Output Graphic View

Groups of service units are often developed or collected together, to form parts of a larger application or new service that are tested and deployed together. A group of JBI service units, along with a description of their relationships and target components, is called a *service assembly*. Such an assembly is deployed into a JBI environment; the JBI implementation is responsible for deploying the individual service units to the appropriate plug-in components.

The service assembly is a package (ZIP file archive) of individual service unit packages (as described above), plus an assembly descriptor file named META-INF/jbi.xml. The descriptor includes data describing where each service unit is to be deployed, as well as connections between the service units to other service units, or external services. An example service assembly is shown graphically in figure 1.

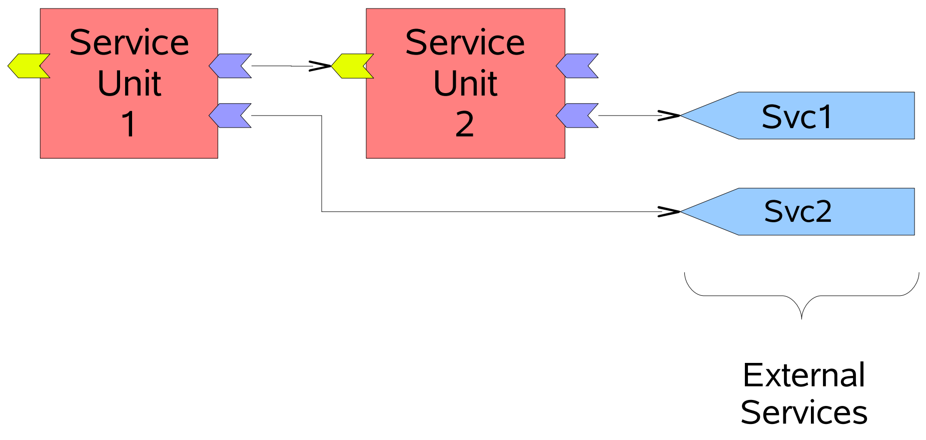

Figure 5: Example Service Assembly

Connections are shown between external service providers (on the right) and the service dependency "pins" shown for service units 1 and 2. Service unit-1's other dependency "pin" is shown connected to the service provided by service unit 2.

Service unit 2 has one unconnected service dependency chevron. This indicates that this dependency is to be resolved dynamically, at run-time, either by the component or the JBI normalized message router.

Connections between provider and consumer pins (yellow and blue chevrons, respectively) must be type correct. That is, the service type (as defined in WSDL) must agree between what the consumer wants, and what the provider provides.

The services provided by both Service Unit 1 and Service Unit 2 are available (as external services) to JBI service consumers, including service units in other service assemblies.

Service-Oriented Integration

Building a service-oriented architecture rarely involves starting from scratch, creating each service that is needed. Instead, existing IT business functions must serve as the basis for many services, and must be integrated into the SOA. Creating a service-oriented architecture out of disparate applications is a relatively new approach to enterprise integration, and is called service-oriented integration (SOI). The advantages of SOA, discussed above, make for a far more flexible, usable (and reusable) integration architecture, where changes can be easily, flexibly, and reliably accommodated.

Building SOA applications involves many integration activities. This section will illustrate, by solution of a simple integration problem using JBI, many of the steps involved in SOI.

Example: Adapter Pattern

A common problem in enterprise integration involves adapting incompatible protocols and message formats such that two entities can interact. This is illustrated in the figure 6. An existing entity can generate request messages, using format "J", and sending them via a particular communications protocol "X". An existing responding entity can receive messages using format "K" via protocol "Y". From an application viewpoint, the two message types convey the same information – they are simply formatted differently.[1]

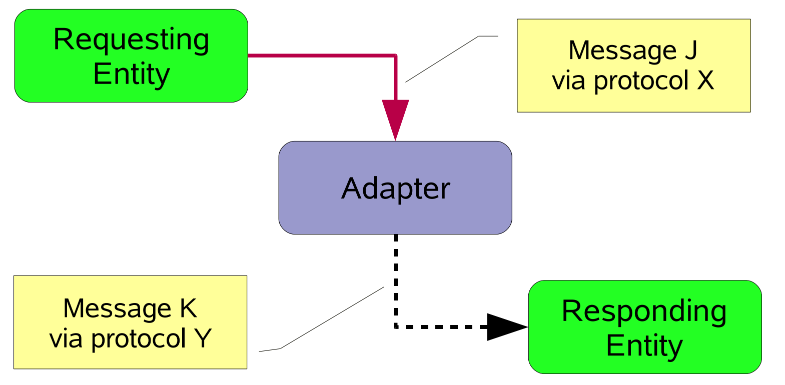

Figure 6: Message Adapter Pattern

The adapter must perform two distinct functions:

- converts messages from format "J" to format "K";
- handle both communications protocols, such that it allows the (converted) message to "hop" from protocol X to protocol Y.

The objective of the adapter is to allow the two entities to interact without modifying them. This is too simple an example, as it stands, to illustrate SOA, but it will show some basic SOI activities. The fuller SOA picture will be discussed in the section *Expanding the Example: Architectural Aspects*, on page 10.

Mapping this standard integration pattern to a JBI implementation is very straightforward. For messaging protocols X and Y we will will assume the presence of suitable binding components that handle those protocols. Let's call them BC-X and BC-Y, for protocols X and Y. These binding components will supply us with the adapter's protocol "hopping" capability.

1 The message conversion problem can be more complex, of course.

For message conversion, we will need a service engine that provides the appropriate conversion technology, which we customize to perform the conversion from message format "J" to "K". For example, since messages in a JBI system are in XML formats, an XSLT transformation service engine would be appropriate. It would be configured to provide the needed conversion service, using a XSLT style sheet designed to convert from format "J" to "K". Let's call that service engine SE-TX. Finally, we'll need a service engine to provide the logic for co-ordinating the actual conversion and protocol hop.

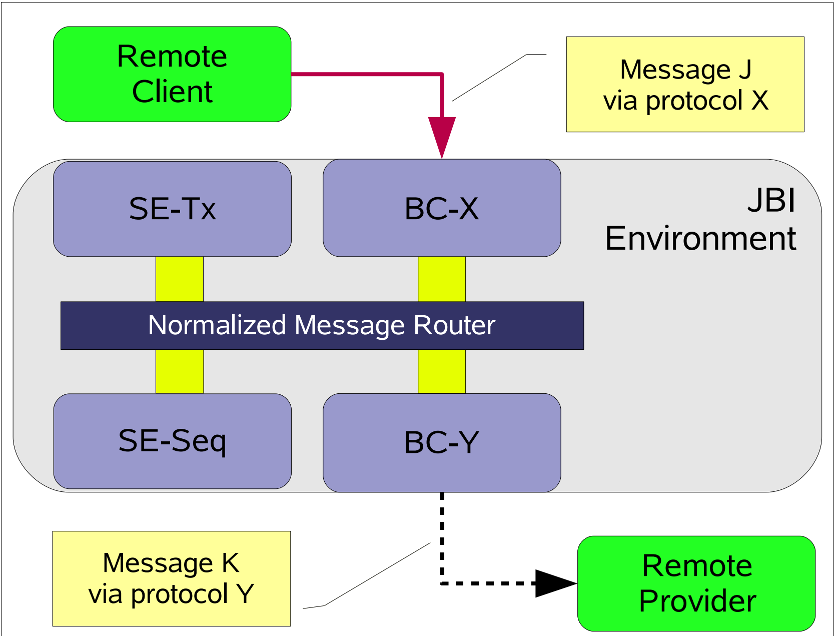

Figure 7: Example JBI Components for Adapter Application

These various plug-in components are pictured in figure 7. BC-X exposes a communications endpoint which the requesting entity (which is a remote client in the JBI services model) uses to submit a request message. The responding entity, which is a remote service provider in the services model) can be accessed through BC-Y.

In the next subsections, we will look at the services provided or used by these JBI components, and then examine how they are composed to provide the needed adapter, following the adapter pattern.

SE-Tx: Transformation Service

The transformation service needed is, in this example, achievable using an XSLT style sheet. If the service is not already available, it must be added to SE-Tx by deploying an appropriate service unit. From the JBI perspective, the service unit will define the following:

- A provided service, named (in this example) http://www.example.com/transforms, operation name "ConvertJtoK". The operation is in-out (request response); the input message is of type "J", while the output message is of type "K".

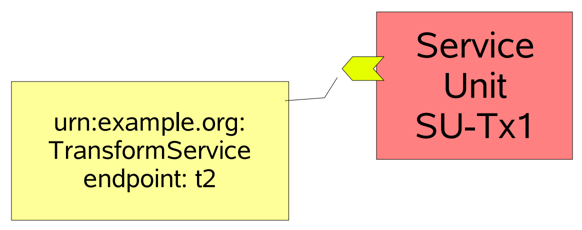

Figure 8: Graphic view of transformation service unit

This is expressed by the META-INF/jbi.xml file included with the service unit's package. The rest of the package is opaque to JBI. For purposes of this example, this would include the XSLT style sheet.

BC-Y: Provider Service Proxy

The remote service provider in this example is accessible using protocol Y, which is known to BC-Y. From the JBI perspective, the responder entity is a service, provided by BC-Y. BC-Y is a proxy for accessing the remote service provider. BC-Y must map service requests from within JBI to appropriate protocol Y messages, sent to the appropriate address.

It is the responsibility of any service provider component to map from the WSDL-based services model used in JBI to the appropriate domain concepts of the component's own realm. In this case the component would map requests for a particular operation of the LoggerService to sending an appropriately formatted message (based on the contents of the NormalizedMessage in the request) to the external service provider.

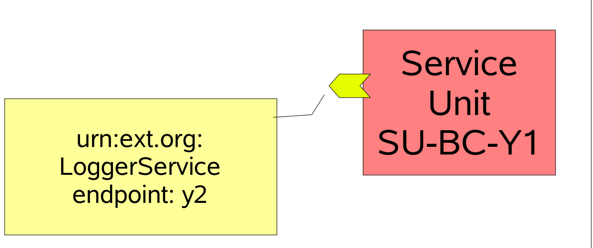

Figure 9: Graphic view of provider service service unit

SE-Seq: Adapter Logic

This component functions as both a service provider and a service consumer. It provides a service that performs the transform-and-forward adapter capability. It does this by consuming two service types: the TransformService, and the LoggerService. These dependencies are expressed in terms of service types; the service unit contains no knowledge of SE-Tx or BC-Y; such knowledge would create coupling between the service units, limiting reuse and flexibility severely.

Note that this component provides a service (of type POLoggerSvc) that can be used by any consumer within the JBI environment.

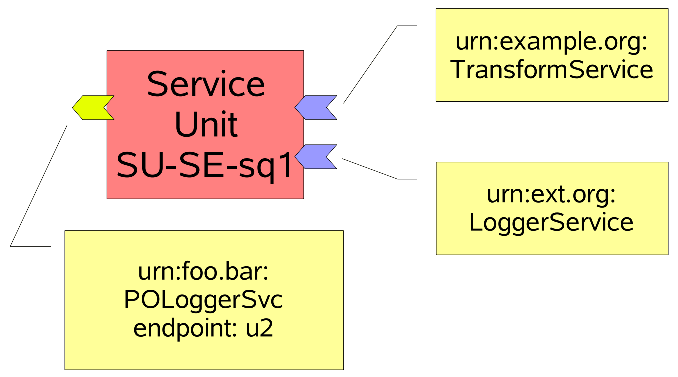
Service Unit SU-SE-sq1
urn:example.org: TransformService
urn:ext.org: LoggerService
urn:foo.bar: POLoggerSvc endpoint: u2

Figure 10: Graphic view of adapter service

BC-X: Client Service Proxy

In order to make the service provided by SE-Seq accessible outside of the JBI environment, a binding component must be configured to function as a proxy for external service consumers. In this example, BC-X must be configured to make the service available (via protocol X) to the external consumer. The component will act as a proxy consumer of the service of type POLoggerSvc.

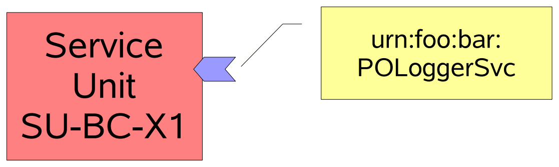

Figure 11: Graphic View of client service proxy

Composition: Putting the Services Together

This division of the adapter pattern leaves us with service units that are defined by their services: the services provided, and the services consumed.[2] By decomposing the problem into separate services, we are creating reusable pieces that can be composed into new applications without modifying the services themselves. We also enable “rewiring” of an application (by switching service providers) without affecting any of the service consumers.

How do we “wire” an application? We do this by creating a service assembly, which contains the needed service units, and metadata defining the connections between service consumer and provider “pins” (the chevrons in the graphics) of the service units or other services external to the service assembly. This is illustrated in figure 12.

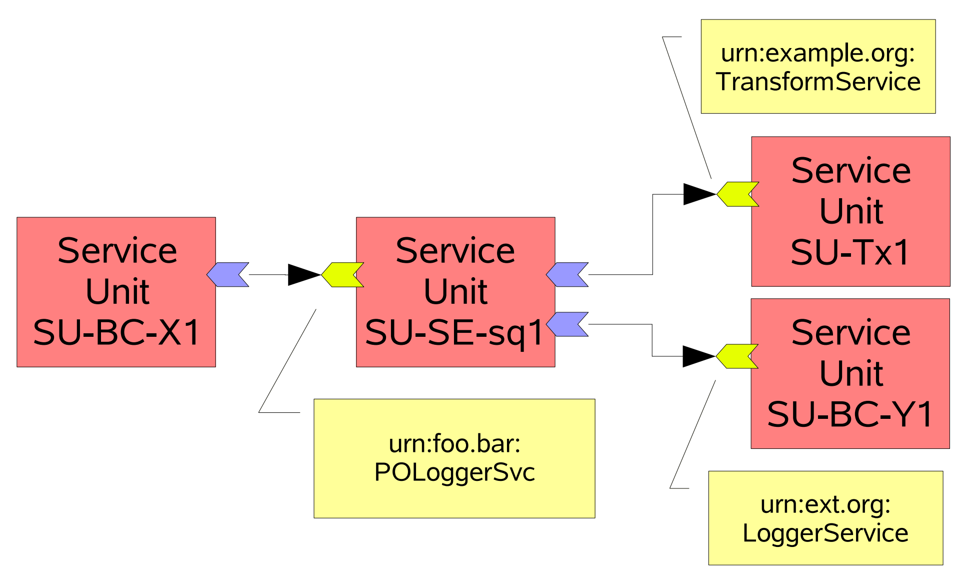

Figure 12: Graphic view of adapter service assembly

Connections (the so-called wires) are made between compatible consumer and provider “pins”. The process of putting together an application, such as this example adapter, is called “composition.” A service unit can be easily recomposed by substituting a compatible service unit. For example, the transformation service unit, SU-Tx1, could be replaced by another service unit that is to be deployed to a different plug-in component that provides a different type of transformation technology. As long as the “pins” remain type compatible (supporting the same WSDL service type), the change of service unit is easily accomplished without affecting the other service units.

Reuse of the services is provided by deployed service units is permitted. This is shown in a service assembly by connecting a consumer “pin” to a particular service endpoint that is not provided within the service assembly.

The Composite Adapter Application at Work

After creating the service unit, it is deployed into the JBI environment. Deployment will cause each service unit within the assembly to be deployed to the appropriate plug-in component. The “wiring” between the service units is used by the JBI implementation's NMR to configure its internal routing tables, ensuring that the declared connections are used at run-time as the components interact.

The sequence of interactions between the plug-in components, as well as the remote consumer and

2 These services (or, more accurately, service types) are the static ones known when the service unit is designed. Dynamic service discovery is also supported by JBI, but this is a more complex form of composite application design than can be explored in this paper.

provider, are illustrated in figure 13. To simplify the diagram, interactions between components, which are always intermediated by the NMR, are shown as lines with double arrow heads.[3]

The sequence begins with the remote consumer sending a request message (REQJ) via protocol X to BC-X. As far as the remote consumer is concerned, that is the extent of the system it deals with: a protocol "endpoint" exposed by BC-X.

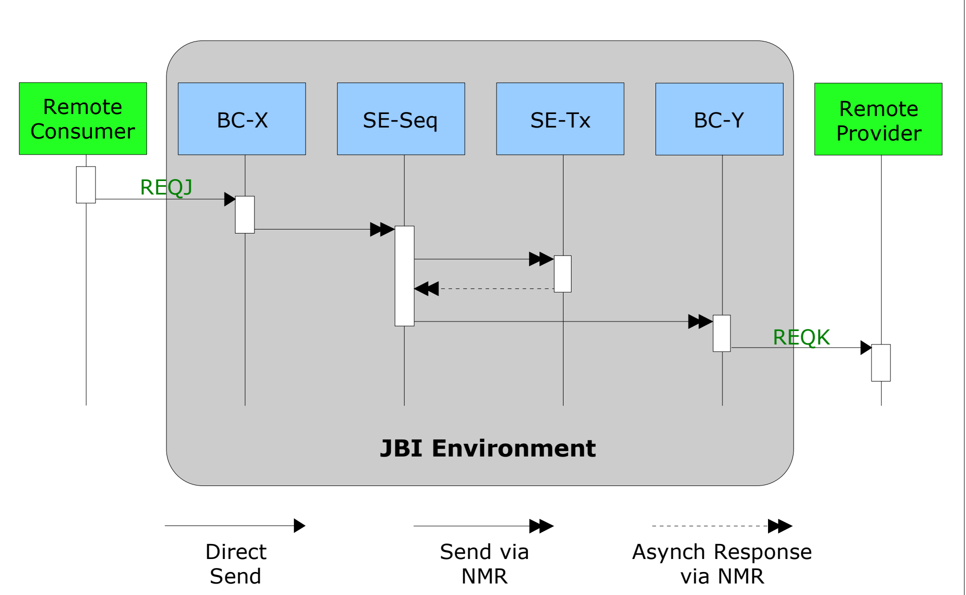

Figure 13: Adapter pattern message sequence diagram

BC-X maps this inbound request to a particular operation of the POLoggerSvc, converting, if necessary, the message to an XML representation. BC-X sends a message containing the normalized (XML) message, addressed to the appropriate service and operation names, to the NMR. The NMR, configured with the connection data provided in the service assembly, will route the request message to SE-Seq.

SE-Seq, receiving this message (and service address), will map the service and operation names of the request to the appropriate logic within the engine, in this case the "process" used to sequence the adapter pattern logic as a simple sequence of service invocations. It performs the following, in sequential order:

1. It creates a new request message, containing the same normalized message as just received. The message is addressed to an appropriate operation of the transformation service (urn:example.org#TransformService). It sends this new request to the NMR, which routes it to the SE-Tx engine.
2. Upon receiving the response from its TransformService invocation, the SE-Seq engine creates a new message containing the response message it just received. The message is addressed to an operation of the LoggerService. The SE-Seq sends this message exchange to the NMR, which routes it to BC-Y.

When SE-TX receives the message sent in step 1 above, it maps the service and operation requested to a particular transformation type (XSLT style sheet). It invokes the transformer, using the transformation type and the received XML document as inputs. When the transformation is complete, it creates a normalized message containing the result. It then sends the result to the NMR, which routes the message to SE-Seq (because SE-Seq originated the request).

When BC-Y receives the message sent in step 2, above, it maps the service and operation requested to a

3 Note that this paper describes the interactions between JBI components abstractly. The actual mechanics of exchanging messages between components are somewhat more complex than described here.

particular messaging protocol endpoint. It transforms the received message into protocol-compliant format, and does whatever other work is needed to comply with protocol Y. The message (REQK) is then sent by BC-Y directly to the external service provider.

Note that the adapter pattern is realized entirely from services, and that the services themselves can be reused in other applications, including the POLoggerSvc provided by SE-Seq. This could be used by a different binding component, in a fashion similar to BC-X, thus providing a new protocol “hop” for the adapter without changing existing services at all. The POLoggerSvc could conceivably be used by more complex business process logic in a workflow service engine. The key enabler is the lack of coupling at the application level between the service provider to the consumer: the provider doesn't “know” why it is being invoked.

Expanding the Example: Architectural Aspects

The above adapter pattern doesn't, in itself, show us much about SOA. Indeed, many integration middleware systems depend on repeating the adapter pattern in a point-to-point fashion, addressing interoperation needs of the all the entities in an enterprise in a pair-wise fashion. This works well for very small numbers of interoperating entities, but encounters significant scaling limits as the number of entities grows. As shown in figure 14, a fully interconnected system of only four entities already involves a significant number of potential connections – N^2 -N, or 12 in this example. Adding one more entity raises the total to 20. An average enterprise with over 50 entities to interconnect will be looking at over 2,500 individual adapters, which is clearly impractical. Of course, fully interconnecting all entities isn't usually required. The rule of thumb in integration practice is that about half the maximum possible number of connections will be needed: (N^2 -N) / 2.

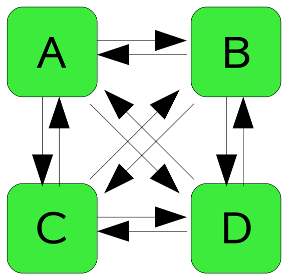

Figure 14: Four Fully Interconnected Entities

How do we address the scalability limits of pair-wise integration? One approach is to limit the number of adapters needed by introducing a common protocol and set of message types. This requires two adapters, as illustrated in figure 15.

The requesting entity is connected to an adapter designed to “canonicalize” the messages it sends, converting message type “J” to the canonical type, “L”. Similarly, the protocol “X” is adapted to the canonical protocol, “Z”.

The responding entity is similarly adapted, but the conversions done oppositely: it functions as a “decanonicalizer”. Its adapter converts the canonical message format “L” to format “Y”, and adapters the canonical protocol “Z” to protocol “Y”.

In a system with but two entities, this pattern serves to introduce extra complexity, and processing, compared

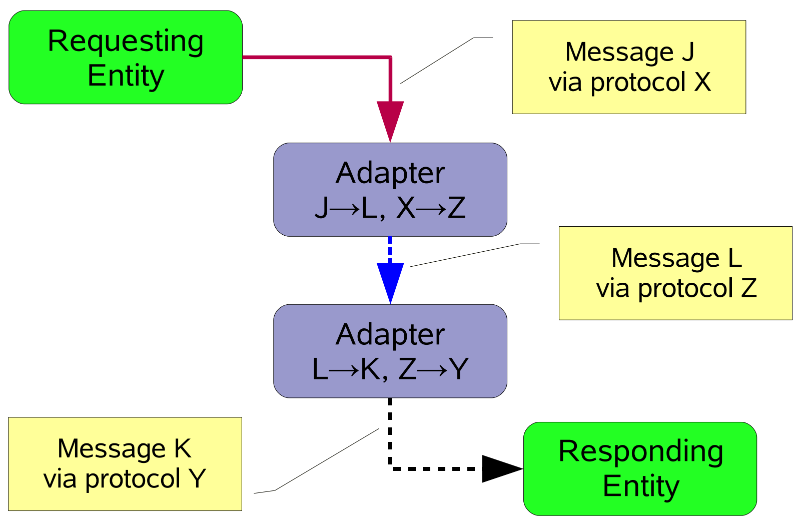

Figure 15: Canonicalizer Adaptation Pattern

to the pattern shown in figure 6. This complexity situation is rapidly reversed, when more interoperating entities are introduced into the system.

A fully interconnected system of four interoperating entities is shown in figure 8. Compared to figure 14, this is much less complex. The number of adapters required is now 2N for full interconnection: the number of adapters is a linear function of the number of entities. Adding a new entity involves only adding (at most) two adapters, rather than adapters for all existing entities in the system.

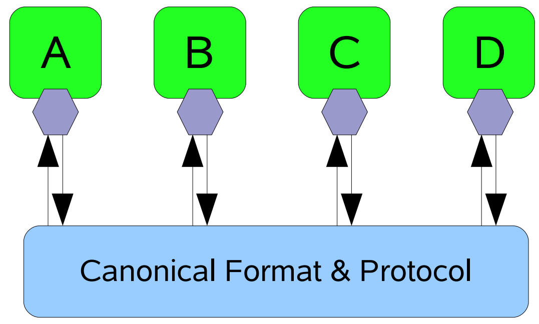

Figure 16: Four Fully Interconnected Entities Using Canonical Adaptation

This pattern illustrates one of the key features of SOA: decoupling. The introduction of a canonical message format and protocol allows the individual entities (and their adapters) to be disassociated from each other. This decoupling limits the scope of changes in the system. Introduction of new entities is one example of such a change. In the directly coupled case the addition of a new entity to a system of N existing entities will require up to 2N new adapters (1N being typical). In the decoupled case, a maximum of two new adapters will be required, for all values of N.

Consider other types of changes: suppose the application represented by entity “A” is upgraded, and some of its message formats are changed. In the directly coupled case, those changes would require updates to up to N-1 different adapters, updates that would have be co-ordinated with the application upgrade. In the decoupled case, only the application and its two adapters need to be updated.

SOA involves more design features than simple decoupling, but it is decoupling that delivers many of the benefits of SOA.

Conclusion

Construction of applications using a service-oriented architecture differs significantly from techniques used in other software architectures. SOA requires that the software developer create services (or integrate existing business functions as services) that are reusable in different applications, including the creation of new services. Each service is designed for such reuse. Reuse is enabled by decoupling the service from the consumer.

Applications are composed from different services. Composite application development is typically accomplished by means of an “orchestration engine”, which invokes (and provides) services according to user logic and engine state. Business process engines are good examples of orchestration engines.

Java Business Integration (JBI) provides a foundation for construction a SOA. It provides for integration of existing business functions as services, and decoupled interaction between service providers and consumers. It provides direct support for composite application creation through the mechanism of JBI service assemblies, which allow applications to be composed directly from the service-based interfaces of JBI service units.

This direct support for composite application construction atop a service-oriented architecture and standards-based messaging infrastructure makes JBI an ideal foundation for constructing service-oriented applications and accomplishing service-oriented integration of existing systems.