Review of SCA and JBI in SOA World

By DMITRI ILKAEV

Abstract: The paper provides a review of the Service Component Architecture (SCA) and Java Business Integration (JBI) as two popular models being introduced in order to overcome the limitations of the existing Web Services APIs. The paper gives a brief overview of SCA and JBI together with their comparison. When evaluating these technologies, we look at them from a technical and vendor perspectives as well as how these models are included into development process

Introduction

As the adoption rate of the Service-Oriented Architecture grows, it becomes more apparent that the level of abstraction, provided by Web Services APIs (the most popular technology for SOA implementation today), for example JAX-RPC in Java or Web Services Extension (WSE) APIs in .Net, is not sufficient for effective SOA implementations. As pointed out in [1]:

- The semantics of these APIs is geared more toward technical aspects of services invocations and SOAP processing, then toward service usage and support.
- The majority of them provide only SOAP over HTTP support , which is not always an optimal transport for SOA implementation.
- The majority of them provide only synchronous and one-way service invocation, which are only a subset of the service interaction styles.
- These APIs are directly exposed to the implementation code which leads to the following:
    - Business implementation code often gets intertwined with the service communications support, which makes its harder to implement, understand, maintain and debug.
    - Any API changes (which usually happen at least once a year) require changes in the business implementation.
- These APIs do not directly support many important service runtime patterns. For example, implementation of dynamic routing requires custom programming and usage of additional APIs (JAX-R in Java) for accessing registry.

In attempt to fix some of the issues of the current APIs sets there are currently attempts underway to raise the level of abstraction through defining SOA programming model (with many elements borrowed from other technologies), which aim to reduce the complexity to which application developers are exposed to when they deal directly with middleware or web services specific APIs. By removing the majority of communications support from the business code and hiding it behind programming model abstraction/implementation this approach facilitates:

- Simplified development of business services

- Simplified assembly and deployment of business solutions built as networks of services
- Increased agility and flexibility
- Protection of business logic assets by shielding from low-level technology change
- Improved testability

The three models currently gaining most popularity for SOA implementations are:

- Windows Communication Foundation programming Model from Microsoft, which attempts to simplify service programming by creating a unified OO model for all service artifacts.
- Java Business Integration (JBI) model from Java Community Process, which address complexities and variabilities of services programming through creation of services abstraction layer in a form of a specialized (service) container.
- Service Components Architecture (SCA) from IBM, BEA, IONA, Oracle, SAP, Siebel, Sybase, etc., is based on the premise that a well-constructed component based architecture with well-defined interfaces and clear-cut component responsibilities can quite justifiable be considered as an SOA.

These programming models attempt to go beyond just service invocations by seamlessly incorporating service orchestration support and many of the patterns required for successful SOA implementation. They also serve as a foundation for implementation of the Enterprise Service Bus. We will leave the Microsoft approach outside the scope of this article and will provide an overview of SCA and JBI models in the next sections.

SCA Review

Service Component Architecture (SCA), see [2] and description of Graham Barber from IBM there, is a set of specifications which describe a model for building applications and systems using a Service-Oriented Architecture. SCA extends and complements prior approaches to implementing services, and SCA builds on open standards such as Web services.

SCA encourages an SOA organization of business application code based on components that implement business logic, which offer their capabilities through service-oriented interfaces called services and which consume functions offered by other components through service-oriented interfaces, called references. SCA divides up the steps in building a service-oriented application into two major parts:

- The **implementation** of components which provide services and consume other services
- The **assembly** of sets of components to build business applications, through the **wiring** of references to services.

SCA emphasizes the decoupling of service implementation and of service assembly from the details of infrastructure capabilities and from the details of the access methods used to invoke services. SCA components operate at a business level and use a minimum of middleware APIs.

SCA supports service ***implementations*** written using any one of many programming languages, both including conventional object-oriented and procedural languages such as

Java, PHP, C++, COBOL; XML-centric languages such as BPEL and XSLT; also declarative languages such as SQL and XQuery. SCA also supports a range of programming styles, including asynchronous and message-oriented styles, in addition to the synchronous call-and-return style.

SCA supports ***bindings*** to a wide range of access mechanisms used to invoke services. These include Web services, Messaging systems and CORBA IIOP. Bindings are handled declaratively and are independent of the implementation code. Infrastructure capabilities, such as Security, Transactions and the use of Reliable Messaging are also handled declaratively and are separated from the implementation code. SCA defines the usage of infrastructure capabilities through the use of ***Policies*** and ***Profiles***, which are designed to simplify the mechanism by which the capabilities are applied to business systems.

SCA also promotes the use of Service Data Objects to represent the business data that forms the parameters and return values of services, providing uniform access to business data to complement the uniform access to business services offered by SCA itself.

The SCA specification is divided into a number of documents, each dealing with a different aspect of SCA. The ***Assembly Model*** deals with the aggregation of ***components*** and the linking of components through wiring using ***composites***. The Assembly Model is independent of implementation language. The Client and Implementation specifications deal with the implementation of services and of service clients -- each implementation language has its own Client and Implementation specification, which describes the SCA model for that language.

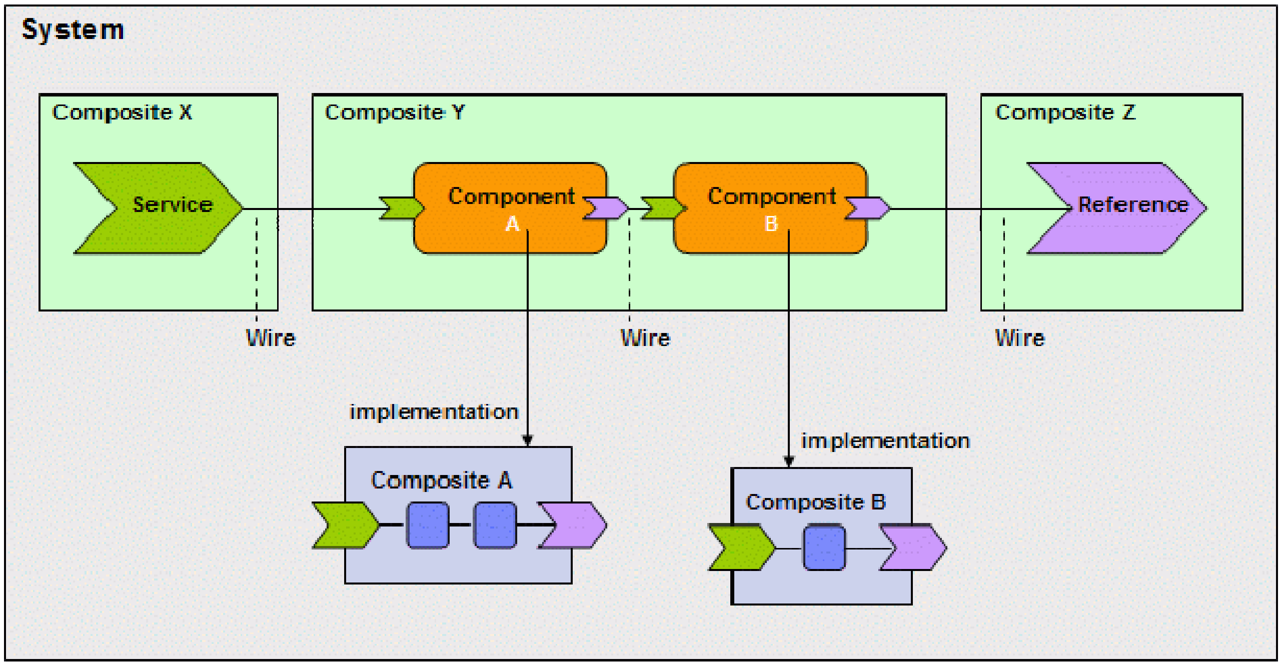


Figure 1. An SCA system built from a series of Composites

Below is the list of the SCA specifications.

Table 1. Final Version 1.0 Specifications

Specification	Date Published
SCA Assembly Model V1.00	March 21 2007
SCA Policy Framework V1.00	March 21 2007
SCA Java Common Annotations and APIs V1.00	March 21 2007
SCA Java Component Implémentation V1.00	March 21 2007
SCA Spring Component Implementation V1.00	March 21 2007
SCA BPEL Client and Implementation V1.00	March 21 2007
SCA C++ Client and Implementation V1.00	March 21 2007
SCA Web Services Binding V1.00	March 21 2007
SCA JMS Binding V1.00	March 21 2007
SCA EJB Session Bean Binding V1.00	March 21 2007

Ronald Barcia and Jeff Brent present a detailed overview of SCA and its implementation using WebSphere platform, see [3-5]. Below are some examples from their review with little or no dependency on the specifics of the WebSphere. The SCA programming model is really about business integration, application composition, and solution assembly, not J2EE application development. An SCA client (which could be J2EE) will typically be external to the process manager and might, for example, use BPEL flows to orchestrate workflow. Web applications co-deployed with BPEL flows can also use the SCA programming model to invoke application-specific functionality and Figure 2 shows an example of an SCA ecosystem.

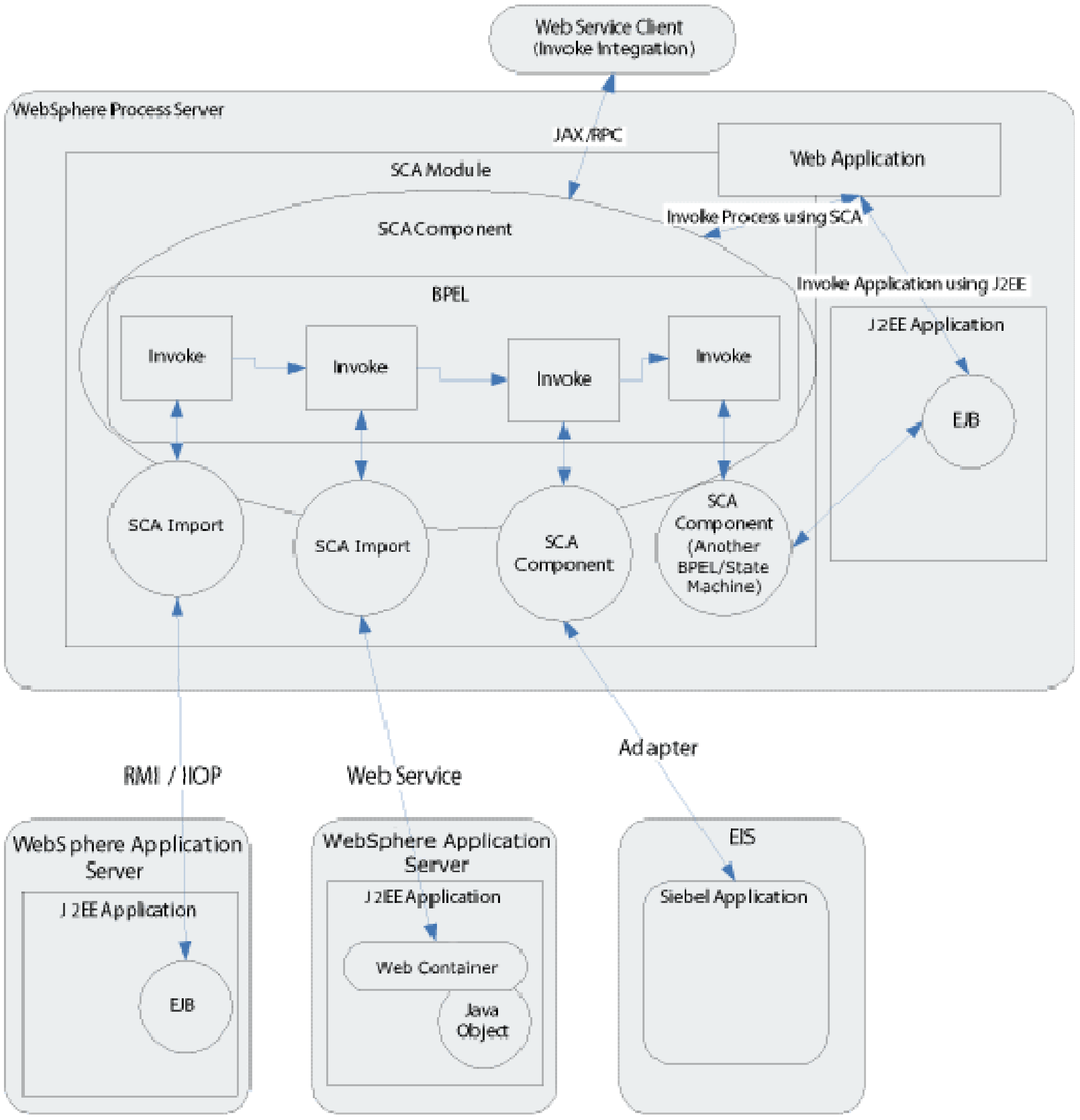


Figure 2. Example of SCA ecosystem

SCA lives in the integration layer. SCA components can use imports to invoke applications outside the SCA runtime. SCA components can be invoked by non-SCA clients using exports. Within the integration layer, SCA components can be composed by defining **references** and using **wires**, which we will focus on in this article. With wires and references, you can define characteristics of the run time invocation at development time; for example, making the invocation synchronous or asynchronous, marking transaction boundaries of the invocation, and so on. These characteristics are read at deployment time and enable the desired run time behavior. Figure 3 illustrates these high level concepts.

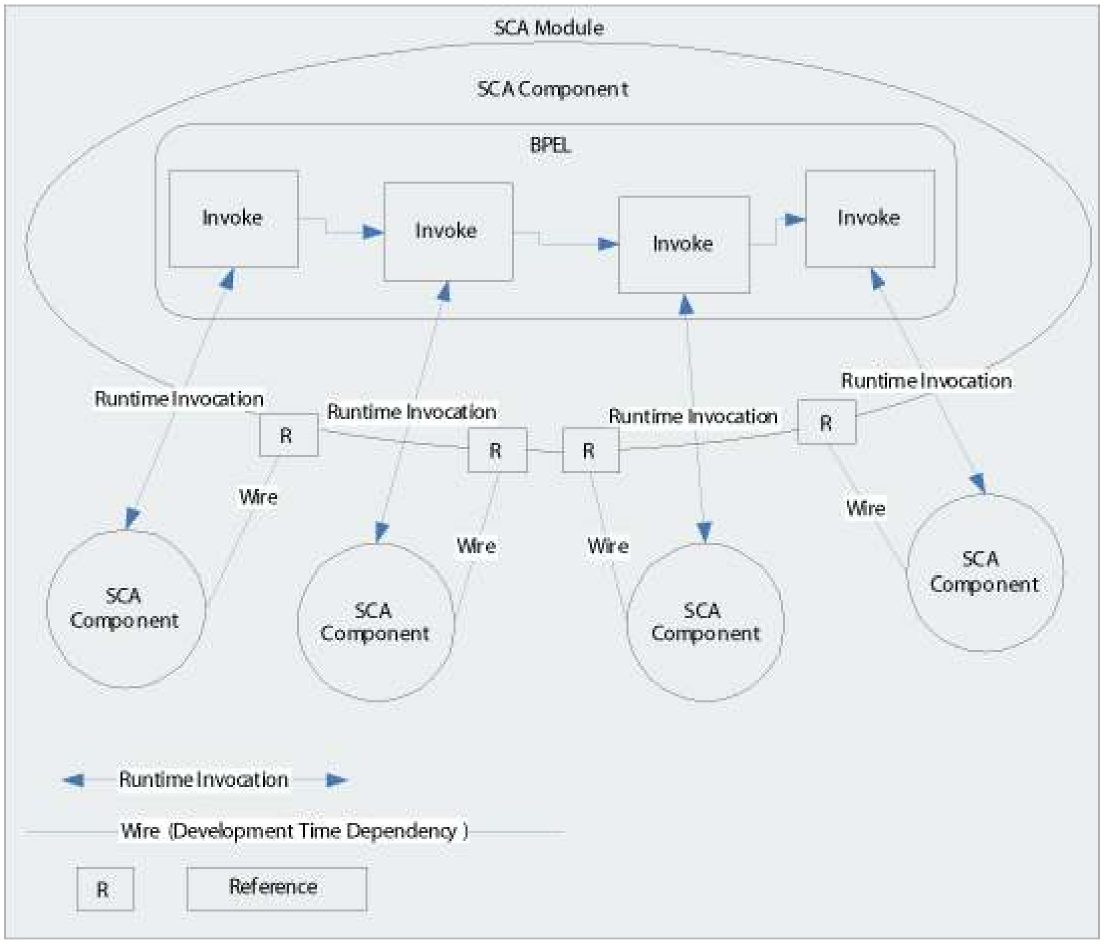


Figure 3. High level view of references and wires

JBI Review

JBI is part of the Java Community Process and defined in JSR 208. JBI is a standard for composing service containers into composite applications. JBI defines an architecture that allows the construction of integration systems from plug-in components, that interoperate through the method of mediated message exchange. The message exchange model is based on WSDL 2.0 or 1.1. Figure 4, see [6], illustrates, at a high level, the JBI concept of plug-in components.

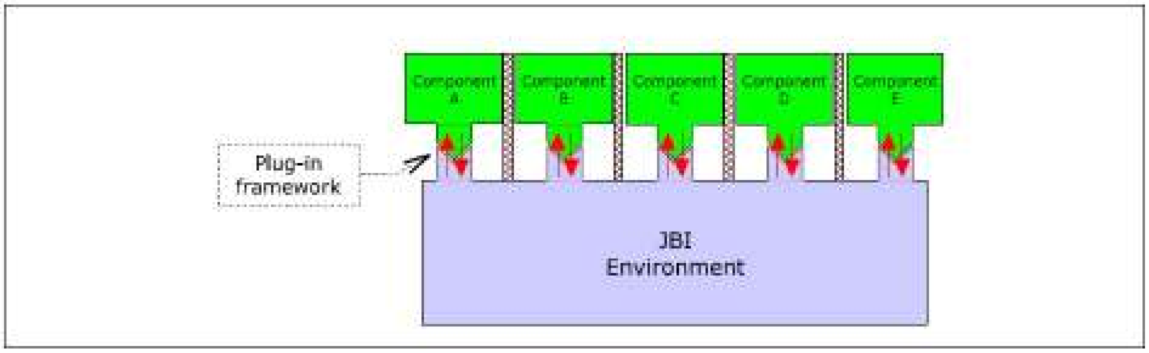


Figure 4 Plug-in Construction of a System

JBI provides specific interfaces for use by a plug-in, while the plug-in provides specific interfaces for use by JBI. Components do not interact with each other directly. Instead, as shown in Figure 5, JBI functions as an intermediary to route messages from component to component.

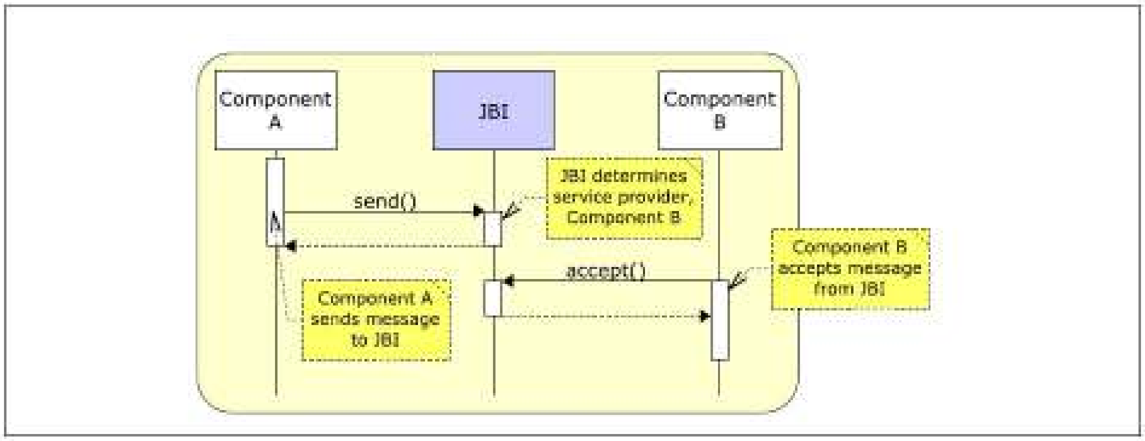

Component A
JBI
Component B
send()
JBI determines service provider, Component B
Component B accepts message from JBI
accept()
Component A sends message to JBI

Figure 5 Mediated Message Exchange: high-level message sequence chart

This separation of the participants in an exchange is key to decoupling service providers from consumers, which is highly desirable in service-oriented architectures generally, and in integration solutions in particular. This preserves flexibility, since consumer and provider are minimally intertwined, and provides a key point for message processing and monitoring in JBI implementations. Note also that this processing is inherently asynchronous; provider and consumer never share a thread. This also helps keep components decoupled.

In this WSDL-based, service-oriented model, JBI plug-in components are responsible for providing and consuming services. By providing a service, a component is making available a function or functions that can be consumed by other components (or even itself). Such functions are modeled as WSDL 2.0 operations, which involve the exchange of one or more messages. A set of four WSDL-defined, basic message exchange patterns (MEPs) crisply defines the sequence of messages allowed during execution of an operation. This shared understanding, between consumer and provider components, of the message exchange pattern is the foundation of interoperability of such components in JBI.

The services provided by components (if any) are described to JBI by the component, using WSDL 1.1 or 2.0. This provides an abstract, technology-neutral model of services using XML-based message exchanges. WSDL also provides a mechanism for declaring additional service metadata of interest to service consumers and JBI itself. Components can query JBI for the for the WSDL describing available services.
As shown in Figure 4, JBI components plug into what is termed the JBI framework. It is expected that such components will be made available to end users from third parties, particularly where common or standardized functions are needed in typical integration problems.
The components are divided into two distinct types:
•**Service Engine (SE)**. SEs provide business logic and transformation services to other components, as well as consume such services. SEs can integrate Java-based applications (and other resources), or applications with available Java APIs.
•**Binding Component (BC)**. BCs provide connectivity to services external to a JBI environment. This can involve communications protocols, or services provided by Enterprise Information Systems (EIS resources). BCs can integrate applications (and other resources) that use remote access technology that is not available directly in Java.
Service engines and binding components can function as service providers, consumers, or both. (The distinction between SEs and BCs is purely pragmatic, but is based on sound architectural principles. The separation of business (and processing) logical from communications logic reduces implementation complexity, and increases flexibility.)
In addition to a messaging system designed to promote component interoperability, JBI defines a management structure, based on Java Management eXtensions (JMX). JBI provides standard mechanisms for:

- Installing components.
- Managing a component's life cycle (stop/start etc.)
- Deploying service artifacts to components.

A top-level view of the JBI architecture is shown in the following figure.

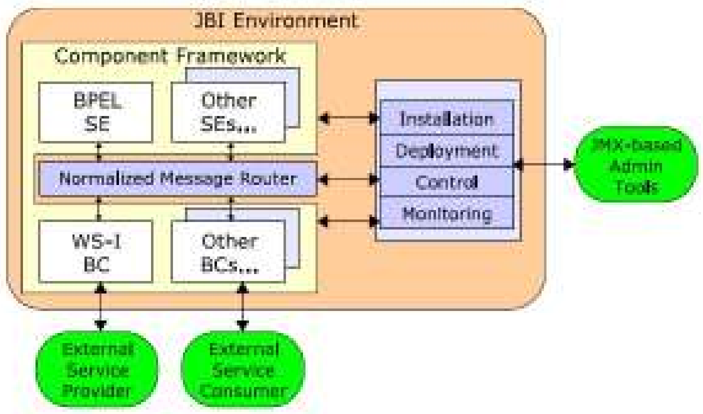


Figure 6. Top level view of the JBI architecture

The JBI environment exists within a single JVM. External to the JBI environment are service consumers and providers, representing the external entities that are to be integrated by JBI. These external entities can use a wide variety of technologies to communicate with Binding Components in the JBI environment. Service Engines are essentially standard containers for hosting WSDL-defined service providers and service consumers that are internal to the JBI environment.

Figure 6 above shows the major architectural components of JBI. This collection of components is called the JBI Environment. Within each JBI environment, there is a collection of services which will provide operational support. Key to this set of services is the Normalized Message router (NMR) which provides the mediated message exchange infrastructure, allowing components to interoperate. In addition, JBI defines a framework which provides the pluggable infrastructure for adding Service Engines (SEs), and protocol Binding Components (BCs). These are depicted within the yellow, C-shaped framework polygon.

The right-hand side of the JBI environment depicts the management features of JBI. JBI defines a standard set of JMX-based controls to allow external administrative tools (shown on the far right) to perform a variety of system administrative tasks, as well as administer the components themselves.

The core message exchange concept implements WSDL messaging, as discussed above. Service requests are generated by consumer components, routed by the NMR, and delivered to a provider component. For example, the BPEL SE may generate a request, which happens to be provided by the external service provider connected to the WS-I BC. The NMR will route the request to the WS-I binding. The SE in this case is a service consumer, and the BC a provider.

All provided services are exposed as WSDL-described services (end points, specifically). Services provided by SEs are described as end points in the same fashion as services provided by BCs (actually by external service providers). This provides a consistent model of service provision, without regard to location. Service consumers may identify needed services by WSDL service name, not by end point address. This decouples the consumer from the provider, and allows the NMR to select the appropriate provider. Service consumers may also identify services by resolving Endpoint References. For example, this allows a JBI component to resolve a service callback Endpoint Reference which was sent in a message.

Aside from the component framework and the NMR, the remainder of the JBI environment provides the infrastructure for life cycle management, environmental inspection, administration, and reconfiguration. These provide a predictable environment for reliable operations.

SCA and JBI comparison

Based on the above sections, JBI is all about the standardization of integration products (mainly ESBs, Enterprise Service Bus). JBI specifies an architecture and an API that should be used when developing integration products. It is a really important standard for the integration market, because it opens up integration solutions and makes it possible for different vendors to collaborate.

SCA, on the other hand, is a specification that defines how developers should write services in a service -oriented architecture. It's like a general WSDL, but also much more. It defines how a service should be described and how it relates to other services. In this way we can define services independent of the technology used to implement it.

Talking about back-end integration in Enterprise Java systems Jean-Jacques Dubray [7] described several solutions: invocation, mediation and activation, see the picture below.

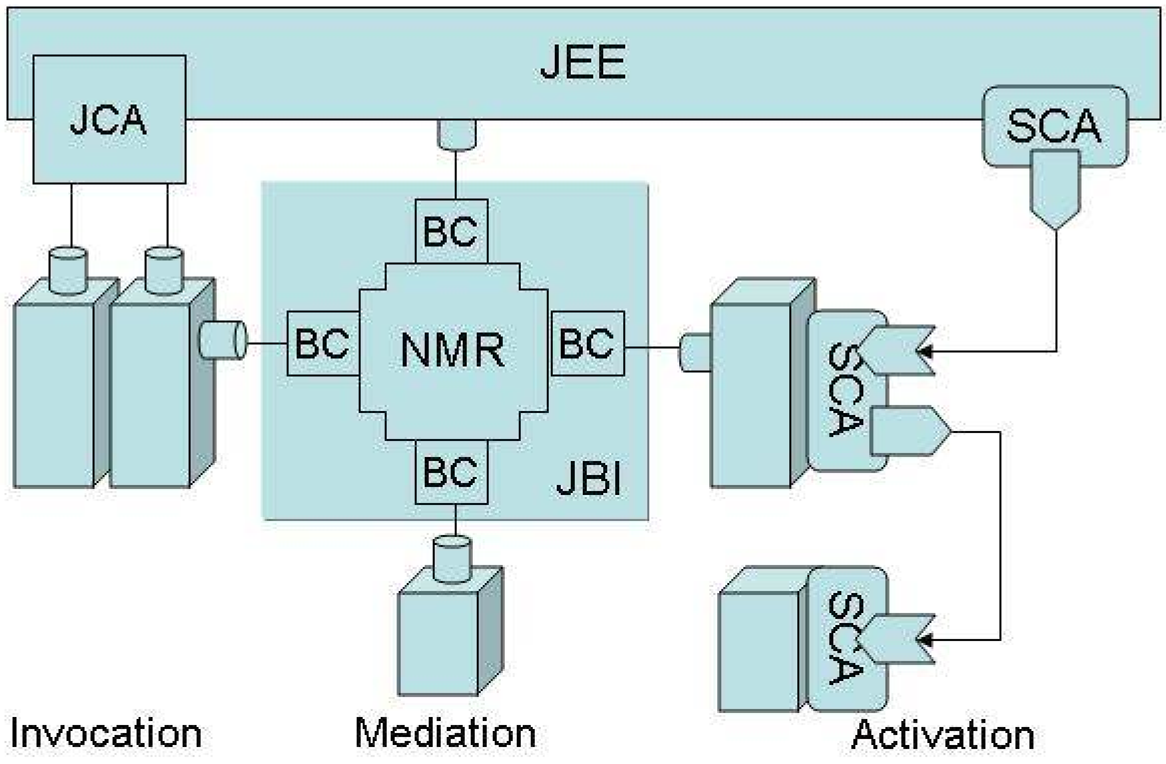


Figure 7. Approaches to develop back-end integration

Invocation has been the most commonly used approach used to date through technologies such as JCA, JAX-WS or WS-IF which are well aligned to support the request/response model but fail to support complex long running integration scenarios associated to particular application states. This is precisely one of the problems in JEE today because back-end integration does not always fit well the synchronous request/response model.

Mediation, is not new, and has been applied through proprietary products and frameworks for a decade. Last summer the JCP published an API to build standardized mediation infrastructure: the JBI specification. JBI is based on the "mediated message exchange" pattern: when a component sends a message to another component, this interchange is mediated by the JBI infrastructure, namely the NMR or Normalized Message Router. The NMR functionality can be augmented by Service Engines (such as transformation or orchestration engines) to facilitate the mediation with back-end systems. However, centrally coordinated architectures, such as JBI, have historically always struggled with the problem of "who manages the central infrastructure". The problem is most acute in B2B scenarios where most companies don't want to incur the cost of "mediating" message interchanges unless there is some value add. Actually, the JBI specification explicitly excludes from its scope the normalization of the interactions between two JBI instances. These type of interactions happen behind binding components in a completely proprietary way. This restriction greatly compromises the composition capabilities of JBI instances. Hence, JBI is well suited to solve small and local integration problems.

Activation is a relatively new approach to the problem. It consists of producing components which can be accessed via different middleware, synchronously or asynchronously. Activation maximizes the autonomy of the components themselves and their ability to be composed with other components. In particular, this means that the business logic implemented by a component cannot depend or rely on any specific middleware facilities. This is the component model proposed by SCA. In SCA, activation applies either at the component level or at the module level. A module is an assembly of components running in the same process.

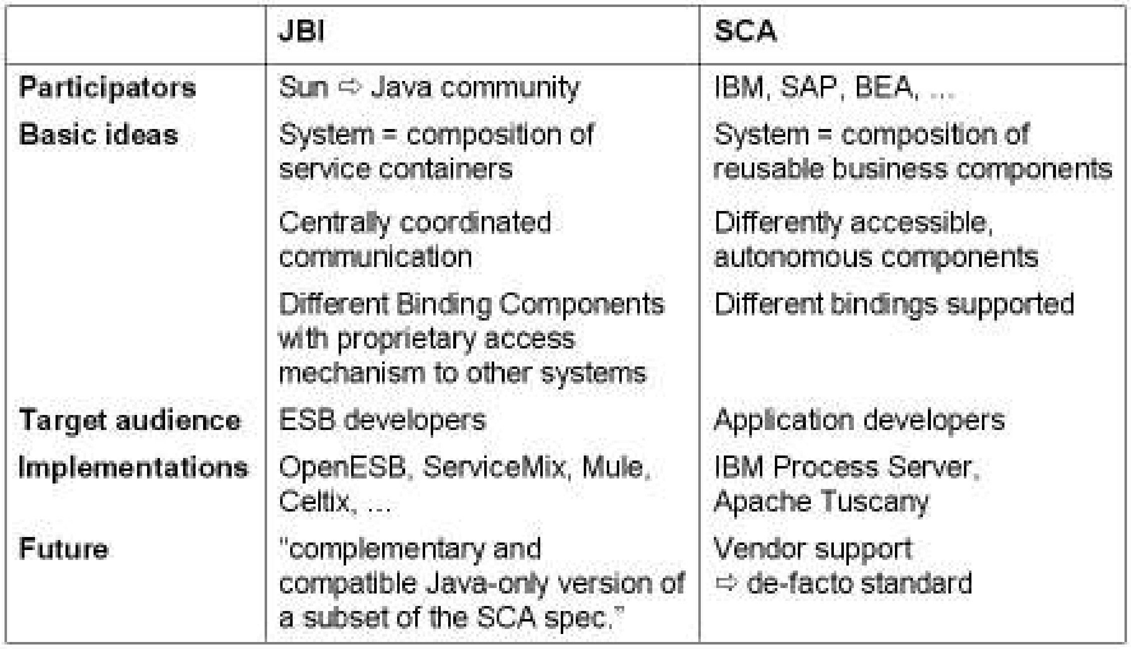
SCA enables arbitrary topologies of systems, supporting synchronous and asynchronous, client/server, peer-to-peer, long running or short lived interactions. SCA does not make any assumption about company boundaries and enables exposing a system as a component participating in another system, each of which having different managing authorities, i.e. company boundaries may be defined or shifted arbitrarily across an SCA system. SCA is well suited to solve any integration problem in particular the most complex ones, including the ones solved by mediation and invocation infrastructures. In many ways, SCA can be viewed as a decentralized mediation infrastructure where mediation happens either at the provider or the consumer side, without necessarily involving an intermediary.

To further understand the differences between the 3 approaches, let's look at how a "connected system" is assembled, i.e. how the wiring is defined and enacted in each approach. In an invocation based infrastructure, the wiring is usually defined via a "connection string containing the end point reference and some credential. In a mediation based infrastructure, the connected system is defined via a configuration file that contains the specification of a particular assembly of components, routing rules, etc. This specification is consumed by the JBI infrastructure (NMR, Binding Components and Service Components). However each binding component retains its own proprietary mechanism to specify wiring to the service provider or consumer behind the binding component. In SCA, there is no central coordinator and an assembly of components is deployed to each component type, which activate components (instances of component types) for each unit of work being performed.

SCA offers a new integration model together with a new application model, i.e a new way to build applications as an assembly of autonomous software agents, exposing service interfaces. We can expect that Java EE and SCA will coexist offering a complementary application model while JBI will be used in traditional Enterprise Application Integration scenario.

Dustin Lange in his Service Component Architecture presentation with the reference to Forrester [8] gives the following comparison between JBI and SCA

Table 2. SCA in comparison



	JBI	SCA
Participators	Sun ⇨ Java community	IBM, SAP, BEA, ...
Basic ideas	System = composition of service containers	System = composition of reusable business components
	Centrally coordinated communication	Differently accessible, autonomous components
	Different Binding Components with proprietary access mechanism to other systems	Different bindings supported
Target audience	ESB developers	Application developers
Implementations	OpenESB, ServiceMix, Mule, Celtix, ...	IBM Process Server, Apache Tuscany
Future	"complementary and compatible Java-only version of a subset of the SCA spec."	Vendor support ⇨ de-facto standard

Analyzing SCA positioning at SOA landscape mainly against the enterprise service bus (ESB) and Java Business Integration (JBI), Daniel Rubio [9] had looked a technical and vendor levels.

At a technical level, SCA differs from an ESB and JBI -- the latter of which is a standardized Java ESB specification -- because both try to solve different problems. While an ESB is charged with bridging the disparities that may surge within existing systems through a bus-like architecture, transforming and adapting requests via services, SCA's role is to offer this same service flexibility, but through a newly-minted application architecture. In this sense, SCA is more keen on being used as a fresh approach to building applications in the context of an SOA than a mediating middleware like an ESB to bring existing applications to participate in an SOA.

At a vendor level, while it may be a question of product line vision or simple coincidence, many SCA supporters who are also involved in the Java space have opted not to participate in the JBI approach, which puts into question not only the need for JBI/ESB, but more importantly the role SCA will play in the evolving battle for what approach is better suited to form the basis of an SOA.

References

1. Boris Lublinsky SOA Programming Models http://www.InfoQ.com

2. http://www.osoa.org
3. Ronald Barcia, Jeff Brent IBM WebSphere Developer Technical Journal: Building SOA solutions with the Service Component Architecture -- Part 1 Oh great, another programming model? http://www-128.ibm.com/developerworks/websphere/techjournal/0510_brent/0510_brent.html
4. Ronald Barcia, Jeff Brent IBM WebSphere Developer Technical Journal: Building SOA solutions with the Service Component Architecture -- Part 2 Assembling SCA components http://www-128.ibm.com/developerworks/websphere/techjournal/0512_barcia/0512_barcia.html
5. Ronald Barcia, Jeff Brent IBM WebSphere Developer Technical Journal: Building SOA solutions with the Service Component Architecture -- Part 3 Integrating SCA modules with imports and exports http://www-128.ibm.com/developerworks/websphere/techjournal/0602_barcia/0602_barcia.html
6. Java™ Business Integration (JBI) 1.0 Final Release. August 17, 2005
7. Jean-Jacques Dubray Comparing SCA, Java EE and JBI https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/2824
8. Forrester on SDO/SCA versus JBI http://www.codefutures.com/weblog/corporate/archives/2006/04/forrester_on_sdo_sca_versus_jbi.html
9. Daniel Rubio A primer on Service Component Architecture http://searchwebservices.techtarget.com/tip/0,289483,sid26_gci1191428,00.html