

Copyright © 2006 Mastermind Media Corp



www.mastermindmedia.com

The State of Systems Integration

By Luvisia M. Molenje

Computer science has never walked the straight and narrow; everything is variable and subject to change from hardware, operating systems, languages and architectures, everything is variable and constantly changing. A side effect is that many innovations that are considered groundbreaking and revolutionary routinely become antiquated and obsolete. Currently, system integration is going through an industry galvanizing paradigm shift. This article is meant to clear up the state of the industry.

The biggest change to system integration in the last ten years has been the advent of eXtensible Markup Language (XML) as the base communication language. In an unlikely coup, lowly marked-up text files supplanted more technically sophisticated object distribution. It is telling that the acronym *Simple Object Access Protocol* (SOAP) has the word object in the name since it admittedly has very little to do with objects. The naming is an acknowledgment to its leading rivals at the time of inception, *Common Object Request Broker Architecture* (CORBA) and *Distributed Component Object Model* (DCOM). It is important to note that SOAP is built on XML.

CORBA and DCOM enabled distributed objects that could provide services. This was the introduction of the *service access layer*. Later J2EE combined the service access layer with a *data access layer* and replaced the brokers of CORBA fame with a container, allowing asynchronous messaging with other layers. This paradigm took the three-tiered architecture to the now loosely coupled n-tier architecture.

It is not necessary to have a distributed architecture to perform system integration. Integration can be primitive in its implementation. For a long time, marked-up text loaded onto floppy disks and walked between systems was a typical integration solution (the fabled sneakernets). Distributed services make integration far simpler. A service can have the sole responsibility of getting data from a foreign system to a centralized location to be shared with the rest of the integrated applications. This hub and spoke approach to integration was the thrust behind Enterprise Application Integration (EAI). Early distributed services communicated at the program language level. The services were replaced with *adapters* that translated data into native languages, creating a more flexible hub known as an EAI broker.

XML untethered the medium of communication from the hub. XML distributed services could communicate with one another directly. A common messaging system was necessary to access adapters from anywhere in the enterprise. Enter *message-oriented middleware* (MOM), which provided *queues* that stored messages similar to answering machines. Programs obtained messages from the queues, processed them asynchronously, and then replied to the appropriate queue. *Java Messaging Service Queues* (JMSQ) provided a messaging system that supported a complete range of messaging architectures, including *publisher-subscriber*, *forward-and-forget*, and *request-reply*. There are other popular message queue services such as MQSeries and MSMQ.

This brings us to SOAP. There are two styles of SOAP, *document* and *remote procedure call* (RPC). These two styles are incompatible. Meaning if RPC is requested against a document style service or vice versa, the call will fail. RPC is synchronous, the calling application must wait for a response from the service before control is returned to the thread that initiated the request; the reason is that RPC actually starts a process on the application server. Document style can be asynchronous since it only delivers a message (an asynchronous message queue).

Technically, any service listening on an IP address can be a web service. For all intents and purposes, web services mean SOAP services. The most defining feature of SOAP is the Web Services Definition Language (WSDL). A WSDL is an XML file that contains information about the process, including the channel it uses. The WSDL is imported and compiled into a calling application (static binding) or looked up at runtime (dynamic binding). RPC must be statically bound. The reason is RPC services represent a procedure on the server, and a change in the parameters must occur in both the server and the *calling application* (client). By contrast, document style XML leaves a message, as long as all of the parameters that the service expects are present and in the proper format. The rest of the message can change arbitrarily.

Dynamic bindings are less brittle than static bindings. Statically bound services are easier to develop and in most cases sufficient. With static binding, a change to the service will cause the calling applications to fail. To realign the calling application and the service, the

new WSDL needs to be imported and the application recompiled and redeployed. With dynamic binding the format of the XML and the channel are looked up at runtime. The two most common directory lookup structures are UDDI and JNDI. UDDI is for web services JNDI is for J2EE, which includes JMSQ. Even with dynamic binding and the document style of web services, a change to the service could cause failures in the calling application, but there are fewer places for error.

Outside of facilitating messages using WSDLs, SOAP resolves two weaknesses of marked-up text files over message queues. The first weakness is a developer has to create an event listener to receive and parse the file, and then process the input. SOAP provides a template for invoking procedures remotely, complete with type resolution. With SOAP it is possible to expose methods without having to do the dirty work of parsing inbound XML or producing XML representations of the outbound message. The second weakness is that message queues require a message server. By comparison, SOAP generally requires a web application server that uses HTTP, although SOAP can also be used over queues. HTTP is the most established web protocol. Even though considerable thought and attention has been put into securing JMSQ, it is not practical to support two different web protocols. This is particularly true when the services are going to be exposed to external organizations.

Section II:

Our story takes a dramatic turn with the introduction of the *Enterprise Service Bus* (ESB). As mentioned before, one of the main weaknesses of communicating with XML over queues is that an event handler has to be written for each request and the inbound files must be parsed before any processing can begin. The ESB provides a harness for processing inbound requests. An ESB is a way to receive requests over a variety of channels and convert the input into a standard XML redux called a canonical. Regardless of how the message comes in, it is *transformed* into a canonical format and forwarded to the appropriate asynchronous message queue.

An *event listener* is a program, or part of a program, that runs when an event occurs. A *service* is an event listener. The service listens on a communication channel. The service uses input parameters received on the channel for the program that it runs upon request. A service usually returns a response. However, it may simply execute or forward its response to another service. A service is distinguishable from other forms of event listeners because it is distributed and communicates with a network communication channel; otherwise it is simply a procedure.

With an ESB adapters do not communicate with a hub as in EAI, instead they communicate with a service container, a type of event listener repository. As mentioned earlier, the ESB container transforms the input into canonical XML files and sends them to another service. Think of it as the U.N., instead of needing a translator to translate among all languages, each

country has a translator to translate between their language and English (canonical XML). All dialogue is translated into English and sent to the appropriate translator (listener). Each translator receives the message and then translates it into his or her native tongue to be processed accordingly.

The final format of the message is unimportant to the ESB. The ESB simply routes transformed messages to and from canonical format and then routes it onto listening services. For example an ESB may forward the request to a message broker. The message broker may use an adapter to communicate with an ERP solution or a databases adapter (driver) to communicate with a database.

An SOA is a part of an ESB or an ESB is an implementation of an SOA. There are many definitions of SOA, ESB, BPMS and MOM. An ESB can effectively double as a Business Process Management System (BPMS). I prefer to break up the definition to separate each concept. A BPMS is code that sits atop an ESB and takes over the responsibility for *orchestration, coordination and state management*. The logic for a BPMS can be integrated into an ESB. Although an SOA can be an ESB and an ESB can be a BPMS, a BPMS cannot be an SOA. These nuanced distinctions are the source of great confusion.

To say there is much fanfare around SOA is an understatement. The exuberance for SOA is pushing organizations' vision of distributed programming to the limit. There are two broad categories within SOA, event driven services and message driven services. Within these two categories,

there are four types of services: *basic service*, *intermediary services*, *process centric service* and *public enterprise services*.

Event driven SOA relies on synchronous *point-to-point* services. As you recall for SOAP this usually means RPC. Using event driven SOA, it is possible to coordinate services in the calling application. A calling application may be a *front-end* or another service. To manage this coordination, tight agreements between service providers and consumers called *service level agreements* are required. This is because the calling application will be waiting for a response from a third party process, out of its control. By contrast, message driven services retrieve messages from queues asynchronously. The message contains enough information to tell the service what to do. Message driven services are generalized event listeners called *message listeners* or *queue listeners*. Event driven services are specialized listeners simply called event listeners. Both are more typically referred to as services in SOA.

Adapters are also event listeners. Since SOA's popularity has blossomed, it is common to refer to adapters as *adapter services*. Adapters are proxy interfaces between the actual program or service and the calling applications. Adapters may be event or message driven. In the U.N. example, the adapter is the translator. SOAP web services provide SOA with a type of generalized adapter. Web services define the channel the adapter is listening on (where a particular translator can be reached). It also predefines troublesome semantics and syntax that can lead to message misinterpretations.

At the heart of SOA is the principal of *loose coupling*, a principal borrowed from distributed architecture. SOA is not synonymous with web services.

However much of SOA would not be possible without services that can easily integrate together, which is essential for loose coupling. Web services publish specifications in the form of WSDLs. The services are accessible to any application with access to the channel and appropriate credentials. The web service is self-contained with regards to its internal processing; all the calling application is privy to be the inputs and outputs.

Implicit in loosely coupled services is that the calling application has access to all of the required input parameters of the called service. In the loosely coupled framework there is little room for negotiating input parameters or responses between the service and the calling application. If the inputs are negotiable, services are tightly coupled. Tightly coupled services are designed to interact with a particular calling application. Loosely coupled services are designed to perform a task in a business process. For this reason, a variety of stakeholder interests must be considered in the upfront planning.

In a tightly coupled application, such as a client-server application, the front-end interface communicates directly with the data source. It is inconceivable to discuss changes to the front end without coordinating the changes with the back-end data structure and vice-versa. The front end and the back end have a symbiotic relationship. Hence, the label tightly coupled.

In loosely coupled applications, the front end must have a certain amount of a-priori knowledge of the data structure or its surrogate services. However, a change to the underlying data structure can be masked by the service. Also, a change in the front end can result from reusing and re-sequencing the existing services. This distinction is huge and the primary justification for loose coupling.

SECTION III

SOA as a singular concept is too broad to be useful. Once the SOA can be opened, there are as many design decisions as there are with any other architecture, perhaps more. Earlier we touched on the initial decision point from an SOA perspective—should the services be developed to operate synchronously, asynchronously or some combination of both?

The simplest form of an SOA is point-to-point services with centralized core business logic, such as a credit checking service that can be used throughout the enterprise. This type of service is known as a *basic service*, not because the logic is simplistic, but because it does not rely on an SOA framework to operate.

“Service-oriented development gathers the code necessary to create a version of ‘credit check’ that can be shared by all... systems. The service may be a wholly new chunk of software, or it may be a composite application consisting of code from some or all of the systems. Regardless, the composite is wrapped in a complex interface that hides the complexity of the composite.”

Basic services do not have to be synchronous, but synchronous web services are much easier to develop. The

basic service is a powerful tool and a good starting place for an SOA since it can solve a nagging business problem immediately. Plus, the right core business logic can guarantee reuse of the service. Service reusability is the only legitimate reason to select a distributed architecture over other integration options.

Process engines and *process services* are used to handle business logic that falls outside basic services. An SOA is not inherently process centric. It provides building blocks that can be used to create a process centric solution. This is why an SOA has the potential to make *business process reengineering* (BPR) simpler and be more cost effective.

The process engine is one approach. A process engine or process service is a special type of event listener; it can be a web service but it is often a middleware tool or the *service container* itself. A process service spawns a thread for each request it receives and then routes the request to services in a sequence. After routing a request, the thread may listen for the required response. The service can process the response data in numerous ways, including auditing and logging, *transformations*, *enrichment* and *lookups*. The results are forwarded to the next service, until the final step in the process is complete. Each service in this example represents a state transition. A process that relies on an engine to coordinate the framework can monitor the state of the process, as it executes.

Process services are both clients and services in an SOA framework. There are two categories, stateless and stateful. A stateless process service is an *intermediary service*. Its only purpose is

reporting. It can report information from *data centric services*, information about the parts of the system or information about the system as a whole. Stateful process services are known as process-centric services. Process-centric services are more complex and branch into a separate discipline, *transaction process management* (TPM).

A typical and useful example of an intermediary service is a façade. Façades are significant because they provide a view into the basic services that is more useful to a calling application. In a simple example, the calling application sends a request to the façade; the service splits the request into requests specific to the different basic services, and then forwards each the appropriate request. When both responses return, the engine aggregates them and sends consolidated responses back to the requestor. Although basic services are loosely coupled, façades are more tightly coupled to the supported application.

The most significant intermediary service is the router service. A *Content-based router* (CBR) can parse a request and route it to different services based on content. A router service receives routing instructions with enough information to determine the appropriate routing channel and/or forwards the message to that channel. The instructions can be encoded in the format of the message itself. An XML file formatted as a purchase order may need to be routed to the purchase order service. Alternately, the calling application may provide routing instruction, such as *Check_Credit*, which is routed to the account adjustment service. Since the router service exists outside of basic services, it has to be coordinated by a

process service, which is usually an intermediary service.

Itinerary-based routing is a more elaborate form of content-based routing. With itinerary-based routing, instructions can be quite verbose in the form of punch lists or scripts based on a scripting language such as *Business Process Execution Language* (BPEL). These scripts contain routing logic and *transition logic*, which tell the process engine not only where to send data, but what to do while the data is in transit between services. Alternately, the transition logic would have to be programmed into the process engine directly.

Itinerary-based routing creates an alternative to the process service. In the simple façade example from earlier, the intermediary service is programmed in advance to split the request, send it to two services and then wait for the responses and aggregate them. With itinerary-based routing, the service can receive its instructions about what to do with the input as part of the input. The service may forward the request to a splitter service that, as part of its instructions, passes the split file to the proper basic service. The instructions for the basic service state where to forward responses, an aggregator service. Once the aggregator service receives both input messages, it aggregates per its instructions. Also included are instructions where to send the final results. All logging and auditing instructions would be included in the script as well. The benefit to this approach is most appreciable when combined with a Business Process Management System (BPMS) that has a

generic splitter and aggregator service built-in.

A BPMS is a tool to manage business processes through process engines. The management system has built-in means to perform transaction monitoring, transformations and lookups. Abstract icons and templates represent the scripting language that powers a BPMS to express both business rules and application logic. For example, an aggregator icon and a template that shows the splitter-aggregator pattern. The use of icons and templates is meant to be more accessible to business owners, who could design and execute processes with minimal programmer involvement.

SECTION IV:

State is difficult and counterintuitive in distributed systems, SOA in particular. Asynchronous queuing is extremely important for all complex integration. Keeping with the U.N. analogy, if the translator has to translate everything in real time, something may be missed and the event could be lost forever. With asynchronous messaging, a recorded message is left for the translators, who can listen at their own pace, so nothing is lost.

Message recording is an activity separate from message processing, which often supported by the MOM. Although a queue is a repository for messages, it is still considered unreliable. Queues can fail, be flushed or be removed by the wrong party. As an assurance, a sidebar service is set up to save the status of events as they occur, so transactions can be monitored. Reliable queuing entails

placing a message onto a queue and guaranteeing the message will be there when the listener is ready to process it. This style of reliable queuing is called *store-and-forward*.

Using the store-and-forward approach to messaging, messages are saved in a persistent location. For reliability, a confirmation or *acknowledgement* is sent to the calling application. The confirmation does not indicate that the transaction has been processed. Rather, it indicates that the message has been received and stored successfully in persistent storage. A second confirmation is sent from the end-point to indicate it has received the message from the queue. In this manner, traceability from the calling application to the service end-point can be established.

XML databases are becoming a convenient and useful way to store XML documents. In fact, it is a lot easier to retrieve messages by particular element, using Xquery, even though storage and retrieval are far slower. Alternately, the canonical file is stored as a string of data, and requires another criterion by which to locate the file. Typically some form of time stamping is used. This approach is helpful when there is a rough idea of when an event occurred. However, in high volume systems, recovering a particular transaction based on its timestamp can be daunting.

Traceability is extremely important for many transactions, especially those that carry legal implications. However, store-and-forward is not required for traceability. It is required for data integrity, particularly *rollbacks* and *compensating transactions*. Reliable

messaging and the ability to perform corrective actions are necessary from reliable transaction processing. The ability to perform a corrective action requires knowledge of the state of the transaction when the problem occurred.

In databases, reliable transaction processing revolves around synchronized updates. Transaction consistency has a long history in database transaction processing systems. *Atomicity, Consistency, Isolation and Durability (ACID)* is the standard used by all contemporary database management systems. The key concept in ACID is *two-phased commit (2PC)*. In database transaction processing, transaction consistency is needed when two records must either be updated synchronously or reset. In the first phase, the records are locked with a *select-for-update*. Once both items are locked, a second phase request is sent to actualize the update. After both items are updated, the locks are released.

For distributed systems, transaction consistency is not at the record level. Rather it is at the process level. Synchronicity has a more liberal meaning at the process level; it means that a set of tasks must occur in order for the process to succeed. The outcome of a two-phased commit is binary, either both updates succeed or both are undone. Within a process, the first two of five tasks may succeed. If the third task fails, it is not imperative that the entire process be undone or aborted. There may be an alternate task, called a compensating transaction that is only called upon when the third step fails. A failure in step three may mean step two has to be reprocessed using some new

information obtained from step three; ultimately the process may be salvaged.

In case a process rollback has to occur, we need to know what state to rollback to. This can be very complicated in the absence of explicit locking. A process may take hours, days, weeks, and even months or years to complete. It may not be sensible to lock part of the process for that long of a period. If state changes occur while the process is executing, *interrupts* have to be set up to communicate state changes to the live process. An interrupt is a mechanism to stop the process in order to perform a separate operation before proceeding. An interrupt may terminate a running task or service and reset the inputs or it can exist as a compensating transaction and wait until after the task has finished executing.

The solution for overall process management is a *transaction management system (TMS)*. With process engines, interrupts are fairly straightforward. Since processes are running within a thread, the thread serves as a container for the process. As long as the thread is running, the process is alive. With adequate logging, the state of the process can easily be reset if the thread should crash. Furthermore, interrupting a thread is also undemanding. Intermediary services can be set up to communicate the state of the process. The combination of interrupts and status services enable a robust interface into a process called a *dashboard*.

Most BPMS tools provide all of the features of a TMS, including the dashboard. In addition, many have adopted BPEL as the coding language

for the engines. Icons and templates are translated into BPEL; an XML based scripting language. The process engine has a BPEL interpreter. The interpreter reads the instructions at runtime. This means if the process changes, nothing has to be recompiled and/or restarted. Also, one properly configured process engine is sufficient for every process.

A TMS does not require process engines, although process engines make the infrastructure simpler. The alternative is to have to process state contained in the process global XML file (PGX). This can be done with a section of the PGX that contains something similar to a punch-list or a *manifest*. Either the itinerary-based router or content-based router can evaluate the combination of complete/incomplete tasks and compare them to the successful/unsuccessful tasks and make a decision on the next task.

The PGX approach has the advantage of state-related information contained internally. The entire file will have to be logged after each state transition, but only for the purpose of full recovery, in the case of queue or service midstream kinetic failure. Context is fully *encapsulated*, so retrieving the PGX and interpreting its content should provide complete discovery of the process state. If a task has numerous sub-tasks, then that task has to keep track of its own state. The TMS can only roll back the process, not the internal state of the task; those rules have to be encoded in the service itself.

The punch-list is a metaphor, in actuality, any elements of the PGX can be used to glean state and context. The combination of fields that determine

state can be reinterpreted as a binary decision tree. For example, state (A=T, B=T, C=T) leads to state task X(PXG) and Y(PXG), as where (A=F, B=T, C=T) leads to state Z(PXG) only, and (A=T, B=T, C=F) leads to Y(PXG) and B(PXG) again. This is a very simple and arbitrary example, but you can see how such an algorithm could get quite complicated. I am using basic truth logic to indicate success or failure, but in actuality, a successful response can contain failed sub tasks, further complicating the matter. Also, I used the () notation to demonstrate that each task is receiving the PGX as an input. Each task has to determine what elements of the file are relevant for itself or have an intermediary splitter service that filters the input first.

I do not know if my PGX notation is any easier to read than Robert Milner's pi-calculus notation but they both express the same idea. Pi-calculus is far more expressive. In pi-calculus the target task is a channel, which can be confusing since in a real-world design a single channel can be used for success, failure or other.

There are significant differences between the PGX notation and the pi-calculus notation. The former more closely resembles another process transition notation called Petri-net. Within a Petri-net, the document is represented as a token. It can be split but never really broken up. At least not in a way that would lead to two completely different PGX documents in the end of a state transition. The document is the manifestation of the process.

Pi-calculus is loosely derived from the work of Carl Adams who designed a

formal graphic modeling language called a Petri network. Pi-calculus notation looks something like the following:

$$X(\text{resultA}, \text{resultB}, \text{resultC}, \text{var}) = \\ A_ok(\text{resultA}) + \\ B_ok(\text{resultB}).W\langle\text{resultB}\rangle + \\ C_ok(\text{resultC})$$

$W\langle\text{resultB}\rangle$ shows that the results for B are sent on channel W. The rest of the equation reads the results from A, B and C (plus whatever else in the form of var) are received on channel X. This is not the article to further elaborate on the algorithms that underpin service orchestrations.

BPEL is the preeminent scripting language for SOA orchestration and is based on both pi-calc and Petri-net. It is important to mention these mathematical concepts because predictive logic can be employed to avoid *dead paths*. A dead path is a result set that causes the process to hang indefinitely. Paradoxical conditions such as when A is dependent on B and B is dependent on A, or if the subsequent task can never succeed with a set of results from A, B and C. As orchestrated processes become more complex, dead path elimination becomes a major concern.

Conclusion:

Kudos to anyone with a primarily business background that made it through this entire article. I imagine this as the proverbial sipping from the fire hose. I ran through a lot of concepts very quickly and this is by no means the definitive work on any of them. However, I did not “dumb it down” either. Dumbing down, only leads to problems. Hopefully, I was able to clear up any confusion about key subjects

revolving around the new integration landscape.

Since this is already condensed rather than recapped I will further explore the part of the landscape that is the murkiest. This area is the boundary between SOA, ESB, BPMS and in some cases MOMs. Perhaps it is just the fact that discussion of one of these topics naturally lends itself to discussion of the others directly or indirectly. As it is in any verbose articulation of an ESB you will find mention of dashboards and process monitors. These to me suggest a BPMS. BPMS has been relegated to out-of-the-box tools. This overlap is confusing for the sake of discussion. Lines in the sand need to be drawn.

The area of overlap is even broader for SOA and ESB. I would like to see the definition of SOA be limited to the services (even web service) specific portion of an ESB. For example, if your itinerary-based router is not a web service, then it is part of the ESB and not the SOA. If the entire infrastructure (i.e., logging, auditing, routing, etc...) are made up of web services, then it is a pure SOA. If you have say an “adapter service,” that communicates in mediums other than XML, then those fall outside of the SOA.

Admittedly these definitions are arbitrary, but like children that grow up and need their own identity, these concepts have matured. It is time to give them all their own identities. Clarity is the goal, not arbitrary distinction.

MOM:

Message queues, message servers (SOAP as an asynchronous messaging

system is not part of a MOM it is part of an SOA).

BPMS:

Routing programs (that fall outside of web services), dashboards, transaction monitoring systems, and single sign-on enterprise security.

ESB:

Canonical XML services, auditing, logging, lookups, and data enrichment.

SOA:

Everything web-service related. If you build out a MOM and an ESB using web services, it is SOA. If you build out a BPMS using web-services again it is SOA. If you use basic services and a BPMS tool to orchestrate those services, the data access layer, which consists of the core business services and intermediary services, is SOA. However, the orchestration is BPMS.

To many SOA architects, this is heresy. A great deal of effort has been dedicated to explaining that SOA is more than just web services. I am not ignorant of the fact that the word “architecture” has a specific meaning in both English and computer science. So does the word “object.” We all moved passed that to accept SOAP be not object based.

What do we gain from arbitrary distinctions? The answer is, you can build out everything using web services. It is important from a business perspective to know if that is what is being proposed. If you read a proposal about BPMS, SOA, ESB you probably won't know exactly what is on the table. Hopefully after reading this article you will be able to ask the right questions. A standard set of definitions would be

pretty helpful. The technical point as mentioned before is you can mask any program with a web service. At that point call it SOA. Even without universal agreement on the terms, if you can understand the distinctions that I am trying to make, you passed.

Bibliography:

Hammer, Michael. *Beyond Reengineering* (New York: HarperCollins Publishers, Inc., 1996)

Chappell, David. *Enterprise Service Bus* (Sebastopol: O'Reilly Media, Inc., 2004)

Weerawarana, Sanjiva. Cubera, Francisco, Leymann Frank, Storey, Tony and Ferguson, Donald F. *Web Services Platform Architecture* (Upper Saddle River: Pearson Education, Inc., 2005)

Havey, Michael. *Essential Business Process Modeling* (Sebastopol: O'Reilly Media, Inc., 2005)

Juric, Matjaz B. Mathew, Benny and Sarang Poornachandra. *Business Process Execution Language for Web Services: Second Edition* (Birmingham: Pakt Publishing, Ltd., 2006)

Rosenberg, Jothy. Remy, David. *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption* (Indianapolis; Sams Publishing, 2004)

Hohpe, Gregory and Woolf, Bobby. *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions* (Boston: Pearson Education, Inc., 2004)

Krafzig, Dirk. Banke, Karl and Slama, Dirk. *Enterprise SOA: Service Oriented Architecture Best Practices* (Upper Saddle River: Pearson Education, Inc., 2005)

Kaye, Doug. *Loosely Coupled: The Missing Pieces of Web Services* (Marin County: RDS Press, 2003)

Hammer, Michael and Stanton, Steven. *How Process Enterprises Really Work* (Harvard Business Review Publishing, 2001)

Erl, Thomas. *Service Oriented Architecture: A Field Guide to Integrating XML and Web Services* (Upper Saddle River: Pearson Education, Inc., 2004)