Service Oriented Architecture (SOA) and Specialized Messaging Patterns

Table of Contents

Chief Editor:
Duane Nickul
Contributors/Editors:
Laurel Reitman
James Ward
Jack Wilber

1.0 Thesis

The widespread emergence of the Internet in the mid 1990s as a platform for electronic data distribution and the advent of structured information have revolutionized our ability to deliver information to any corner of the world. While the introduction of Extensible Markup Language (XML)[i] as a structured format was a major enabling factor, the promise offered by SOAP based webservices triggered the discovery of architectural patterns that are now known as Service Oriented Architecture (SOA).[ii]

Service Oriented Architecture is an architectural paradigm and discipline that may be used to build infrastructures enabling those with needs (consumers) and those with capabilities (providers) to interact via services across disparate domains of technology and ownership. Services act as the core facilitator of electronic data interchanges yet require additional mechanisms in order to function. Several new trends in the computer industry rely upon SOA as the enabling foundation. These include the automation of Business Process Management (BPM), composite applications (applications that aggregate multiple services to function), and the multitude of new architecture and design patterns generally referred to as **Web 2.0**[iii].

The latter, Web 2.0, is not defined as a static architecture. Web 2.0 can be generally characterized as a common set of architecture and design patterns, which can be implemented in multiple contexts. The list of common patterns includes the Mashup, Collaboration-Participation, Software as a Service (SaaS), Semantic Tagging (folksonomy), and Rich User Experience (also known as Rich Internet Application) patterns among others. These are augmented with themes for software architects such as ***trusting your users*** and ***harnessing collective intelligence***. Most Web 2.0 architecture patterns rely on Service Oriented Architecture in order to function.

When designing Web 2.0 applications based on these patterns, architects often have highly specialized requirements for moving data. Enterprise adoption of these patterns requires special considerations for scalability, flexibility (in terms of multiple message exchange patterns), and the ability to deliver these services to a multitude of disparate consumers. Architects often need to expand data interchanges beyond simple request-response patterns and adopt more robust message exchange patterns, triggered by multiple types of events. As a result, many specialized platforms are evolving to meet these needs.

This white paper discusses specializations for advanced data exchanges within enterprise service oriented environments and illustrates some of the common architectures of these new platforms.

i. The Extensible Markup Language (XML) is a W3C Recommendation - http://www.w3.org/XML/
ii. Service Oriented Architecture is an architectural paradigm expressed as a Reference Model by OASIS at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm
iii. Web 2.0 is defined as a set of Design Patterns in the O'Reilly book Web 2.0 Design Patterns - http://www.amazon.com/Web-2-0-Design-Patterns-entrepreneurs/dp/0596514433

2.0 An Introduction to Service Oriented Architecture

Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains and implemented using various technology stacks. In general, entities (people and organizations) create capabilities to solve or support a solution for the problems they face in the course of their business. It is natural to think of one person's needs being met by capabilities offered by someone else; or, in the world of distributed computing, one computer agent's requirements being met by a computer agent belonging to a different owner. The term owner here may be used to denote different divisions of one business or perhaps unrelated entities in different countries.

There is not necessarily a one-to-one correlation between needs and capabilities; the granularity of needs and capabilities vary from fundamental to complex, and any given need may require a combination of numerous capabilities while any single capability may address more than one need. One perceived value of SOA is that it provides a powerful framework for matching needs and capabilities and for combining capabilities to address those needs by leveraging other capabilities. One capability may be repurposed across a multitude of needs.

SOA is a "view" of architecture that focuses in on services as the action boundaries between the needs and capabilities in a manner conducive to service discovery and repurposing.

2.1 Requirements for SOA

Figure 2-1 shows an example of an information system scenario that could benefit from a migration to SOA. Within one organization, three separate business processes use the same functionality, each encapsulating it within an application. In this scenario, the login function, the ability to change the user name, and the ability to persist it are common tasks implemented redundantly in all three processes. This is a suboptimal situation because the company has paid to implement the same basic functionality three times.

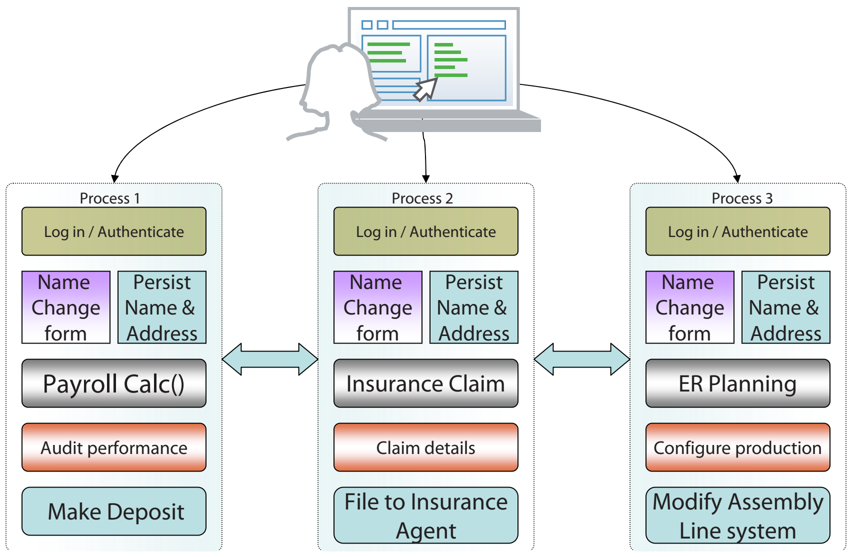

Figure 2.1 – three business processes within one company duplicating functionality

Moreover, such scenarios are highly inefficient and introduce maintenance complexity within IT infrastructures. For example, consider an implementation in which the state of a user is not synchronized across all three processes. In this environment users might have to remember multiple login username/password tokens and manage changes to their profiles in three separate areas. Additionally, if a manager wanted to deny a user access to all three processes, it is likely that three different procedures would be required (one for each of the applications). Corporate IT workers managing such a system would be effectively tripling their work –and spending more for software and hardware systems.

In a more efficient scenario, common tasks would be shared across all three processes. This can be implemented by decoupling the functionality from each process or application and building a standalone authentication and user management application that can be accessed as a service. In such a scenario, the service itself can be repurposed across multiple processes and applications and the company owning it only has to maintain the functionality in one central place. This would be a simple example of Service Oriented Architecture in practice. The resultant IT infrastructure would resemble Figure 2.2.

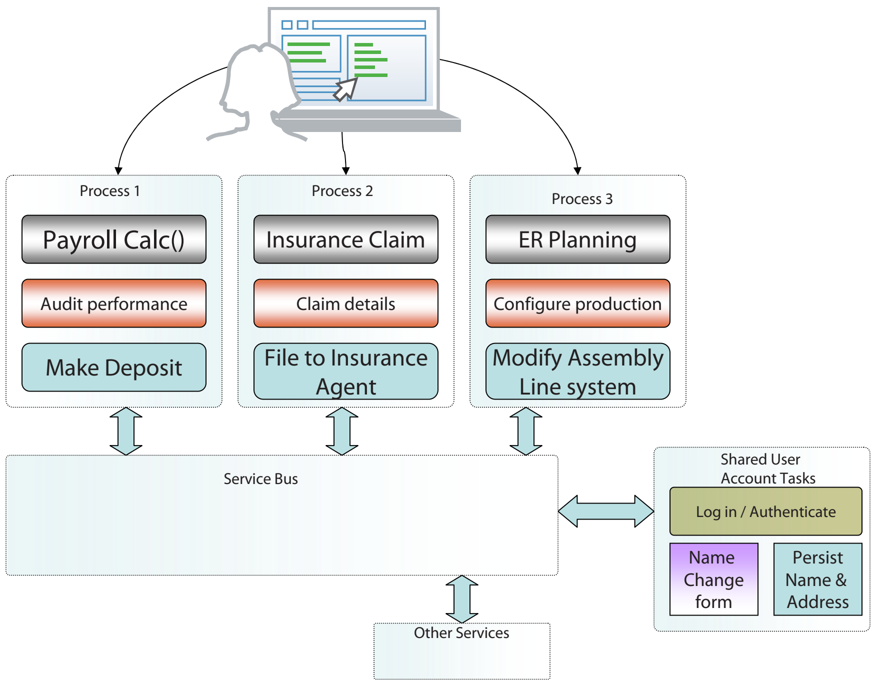

Figure 2.2 – three business processes repurposing one service for common tasks.

In figure 2.2, the shared user account tasks have been separated from each process and implemented in a way that enables other processes to call them as a service. This allows the shared functions to be repurposed across all three processes. The common service bus is really a virtual environment whereby services are made available to all potential consumers on a fabric. This is typically referred to as an ***Enterprise Service Bus (ESB)*** and has a collection of specialized subcomponents including naming and lookup directories, registry-repositories, and service provider interfaces (for connecting capabilities and integrating systems) as well as a standardized collection of standards and protocols to make communications seamless across all connected devices. Advanced ESB vendors have tools that can aggregate services into complex processes and workflows.

In the preceding example of SOA, the complications were relatively minor as the entire infrastructure existed within one domain. In reality, enterprise SOA is much more difficult because services may be deployed across multiple domains of ownership. To make interactions possible, mechanisms have to be present to convey semantics, declare and enforce policies and contracts, the ability to use constraints for data passed in and out of the services as well as expressions for the behavior models of services. The ability to understand both the structure and semantics of data passing between service endpoints is essential for all parties involved.

While most SOA examples are typically shown as a ***request-response*** interaction pattern, more robust exchanges are required. Additionally, modern service platforms also need the flexibility to support these advanced message exchange patterns. Before discussing the platform and reference architecture, this white paper will briefly delve into SOA in more detail.

2.2 A Reference Model for Service Oriented Architecture

As with any other architecture, Service Oriented Architecture can be expressed in a manner that is decoupled from implementation. Software architects generally use standardized conventions for capturing and sharing knowledge. This group of conventions is often referred to as an Architecture Description Language (ADL). There are also several normalized artifacts used to facilitate a shared understanding of the structure of a system, its major components, the relationships between them, and their externally visible properties. This white paper will make use of two special types of these artifacts – a ***Reference Model*** and ***Reference Architecture***.

A Reference Model is an abstract framework for understanding significant entities and relationships between them. It may be used for the further development of more concrete artifacts such as architectures and blueprints. Reference models themselves do not contain a sufficient level of detail sufficient to enable the direct implementation of a system. In the case of a reference model for SOA, the Organization for the Advancement of Structured Information Systems (OASIS) has a standard Reference Model for SOA, shown in Figure 2.3, that is not directly tied to any standards, technologies, or other concrete implementation details.

In order for SOA to be meet these challenges, ***services*** must have accompanying ***service descriptions*** to convey the meaning and real world effects of invoking the service. These descriptions must additionally convey both semantics and syntax for both humans and applications to use.

Each service has an ***interaction model***, which is the externally visible aspects of invoking a service. In this paper, this will be decomposed further to examine the data service aspects of SOA.

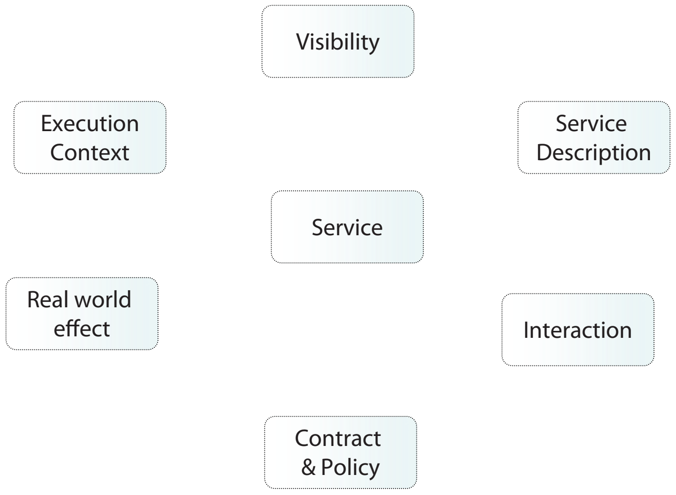

Figure 2.3 – the core OASIS Reference Model for Service Oriented Architecture

Visibility and **Real World Effect** are also key concepts for SOA. Visibility is the capacity for those with needs and those with capabilities to be able to see and interact with each other. This is typically implemented by using a common set of protocols, standards, and technologies across service providers and service consumers. For consumers to determine if they can interact with a specific service, **Service Descriptions** provide declarations of aspects such as functions and technical requirements, related constraints and policies, and mechanisms for access or response. The descriptions must be in a form (or can be transformed to a form) in which their syntax and semantics are widely accessible and understandable. The **execution context** is the set of specific circumstances surrounding any given interaction with a service and may affect how the service is invoked.

Since SOA permits service providers and consumers to interact, it also provides a decision point for any **policies and contracts** that may be in force. The purpose of using a capability is to realize one or more real world effects. At its core, an interaction is “an act” as opposed to “an object" and the result of an interaction is an effect (or a set/series of effects). Real world effects are, then, couched in terms of changes to this shared state. This may specifically mutate the shared state of data in multiple places within an enterprise and beyond.

The concept of **policy** also must be applicable to data represented as documents and policies must persist to protect this data far beyond enterprise walls. This requirement is a logical evolution of the "locked file cabinet" model which has failed many IT organizations in recent years. Policies must be able to persist with the data that is involved with services, wherever the data persists.

A contract is formed when at least one other party to a service oriented interaction adheres to the policies of another. Service contracts may be either short lived or long lived.

2.3 Decomposing the Interaction Model

Whereas visibility introduces the possibilities for matching needs to capabilities (and vice versa), interaction is the act of actually using a capability via the service. Typically mediated by the exchange of messages, an interaction proceeds through a series of information exchanges and invoked actions. There are many facets of interaction; but they are all grounded in a particular execution context – the set of technical and business elements that form a path between those with needs and those with capabilities. Architects building Rich Internet Applications (RIAs), are faced with special considerations when designing their systems from this perspective. The concept of "Mashups" surrounds a model whereby a single client RIA may actually provide a view composed by binding data from multiple sources persisting in multiple domains across many tiers.

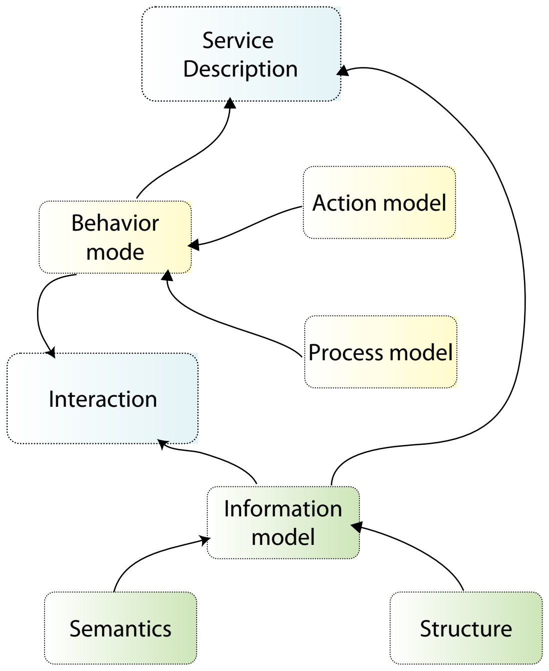

Figure 2.4 – a decomposition of the Interaction Model (courtesy of OASIS Reference Model for SOA)

As depicted in Figure 2.4, the interaction model can be further decomposed into a data model and behavior model. The data model is present in all service instances. Even if the value is "null", the service is still deemed to have a data model. The data models are strongly linked to the behavior models. For example, in a Request-Response behavior model, the corresponding data model would have two components – the input (service Request) data model and the output (service Response) data model. Data models may be further specialized to match the behavior model if it is other than "Request-Response".

The behavior model is decomposable into the action model and the process model. The sequence of messages flowing into and out of the service is captured in the action model while the service's processing of those signals is captured in the processing model. The processing model is potentially confusing as some aspects of it may remain invisible to external entities and its inner working known only to the service provider.

3.0 A Reference Architecture for Service Oriented Architecture

A reference architecture is a more concrete artifact used by architects. Unlike the reference model, it can introduce additional details and concepts to provide a more complete picture for those who may implement a particular class. Reference architectures declare details that would be in all instances of a certain class, much like an abstract constructor class in programming. Each subsequent architecture designed from the reference architecture would be specialized for a specific set of requirements. Reference architectures often introduce concepts such as cardinality, structure, infrastructure, and other types of binary relationship details. Accordingly, reference models do not have service providers and consumers. If they did, then a reference model would have infrastructure (between the two concrete entities) and it would not longer be a model.

The reference model and the reference architecture are intended to be part of a set of guiding artifacts that are used with patterns. Architects can use these artifacts in conjunction with others to compose their own SOA. The relationships are depicted in Figure 3.1.

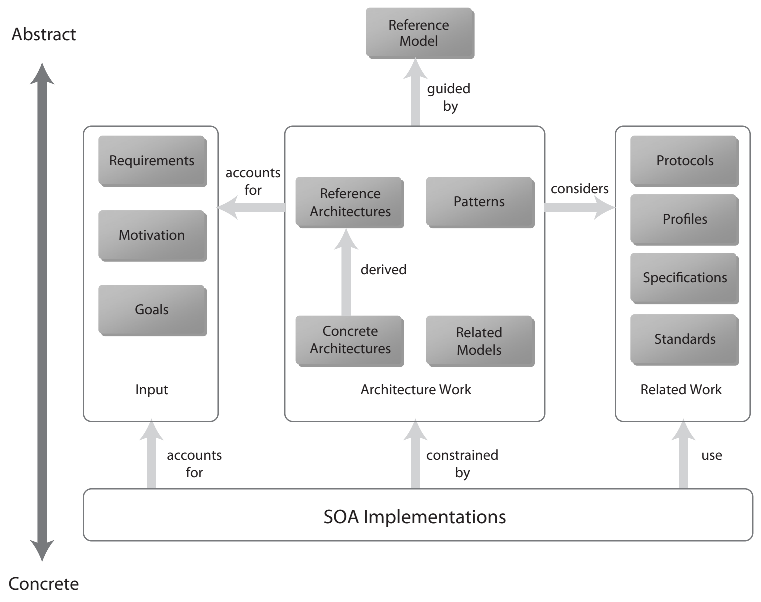

Figure 3.1 – The architectural framework for SOA (Courtesy of OASIS).

The concepts and relationships defined by the reference model are intended to be the basis for describing reference architectures that will define more specific categories of SOA designs. Specifically, these specialized architectures will enable solution patterns to solve particular problems. Concrete architectures may be developed based upon a combination of reference architectures, architectural patterns, and additional requirements, including those imposed by technology environments. Architecture is not done in isolation; it must account for the goals, motivation, and requirements that define the actual problems being addressed. While reference architectures can form the basis of classes of solutions, concrete architectures will define specific solution approaches.

Architects and developers also need to bind their own SOA to concrete standards technologies and protocols at some point. These are typically part of the requirements process. For example, when building a highly efficient client side Mashup application, a developer might opt for the

ActionScript Messaging Format (AMF[iv]) to provide the most efficient communication between remote services and the client .

Neutrality

The reference architecture shown in Figure 3.2 is not tied to any specific technologies, standards, or protocols. In fact, it would be equally applicable to a .NET[v] or J2EE[vi] environment and can be used with either the Web Service family of technologies, plain old XML-RPC (XML – Remote Procedure Call), or a proprietary set of standards. This reference architecture allows developers to make decisions and adopt technologies that are best suited to their specific requirements.

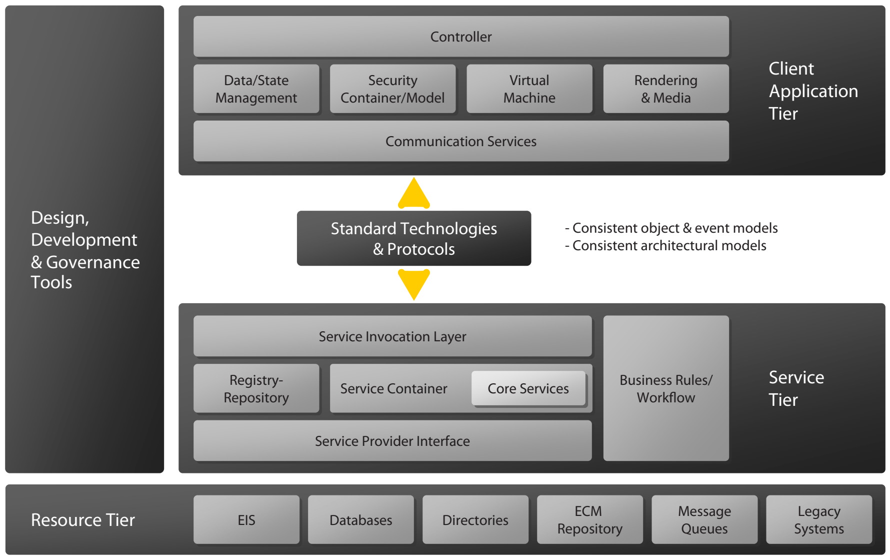

Figure 3.2 – A generic SOA Reference Architecture for implementing core Web 2.0 design patterns (Courtesy of O'Reilly Media)

3.1 Service Tier

The server side component of the reference architecture has a number of commonly used components. The Service Provider Interface is the main integration point whereby service providers connect to capabilities that exist in internal systems in order to expose them as services. These internal applications typically reside in a resource tier, a virtual collection of capabilities that become exposed as services so consumers can access their functionality. Service providers may integrate such capabilities using numerous mechanisms, including using other services. In most cases, an enterprise will use the Application Programmatic Interface (API) of the system as provided by the application vendor.

The **Service Invocation Layer** is where services are invoked. A service may be invoked when an external messages being received or, alternatively, it can be invoked by an internal system or by a non-message based event (such as a time out). It is essential to understand that services may be invoked via messages from multiple sets of standards and protocols working together. Common examples of external service interface endpoints include:

- Asynchronous JavaScript and XML (AJAX[vii]),
- Simple Object Access Protocol (SOAP),

iv. AMF - http://osflash.org/documentation/amf
v. .NET is a trademark and technology from Microsoft - http://msdn2.microsoft.com/en-us/netframework/default.aspx
vi. Java 2 Enterprise Edition is a trademark of Sun Microsystems
vii. Asynchronous JavaScript And XML is described on Wikipedia in more details at http://en.wikipedia.org/wiki/Ajax_(programming)

- XML Remote Procedure Call (XML-RPC),
- a watched folder being polled for content,
- an email endpoint, and
- other REST[viii] style endpoints including plain old HTTP and HTTP/S.

Services may also be invoked by local consumers including environments like J2EE and language specific interfaces (for example - Plain Old Java Objects or POJO's).

Each service invocation is often handed to a new instance of a **service container**. The service container is responsible for handling the service invocation request for its entire lifecycle, until either it reaches a successful conclusion or failed end state. Regardless of its ultimate end state, the service container may also delegate responsibilities for certain aspects of the service's runtime to other services for common tasks. These tasks typically include logging functions, archiving, security, and authentication, among others.

To facilitate orchestration and aggregation of services into processes and composite applications, a **registry-repository** is often used. During the process design phase, the registry-repository provides a single view of all services and related artifacts. The repository provides a persistence mechanism for artifacts during the runtime of processes and workflows. If multiple system actors use and interact with a form, the repository can persist it while allowing access to privileged individuals.

Design, development and governance tools are also commonly used by humans to deploy, monitor, and aggregate multiple services into more complex processes and applications.

3.2 Client Tier

While much attention has been focused on the server side aspects of SOA, less has been written about the new breed of clients evolving for consuming services. The clients have evolved to embrace many common architecture and design patterns discussed in greater detail in the next section. A highly visible example of this is the ability of most modern browsers to subscribe to RSS feeds.

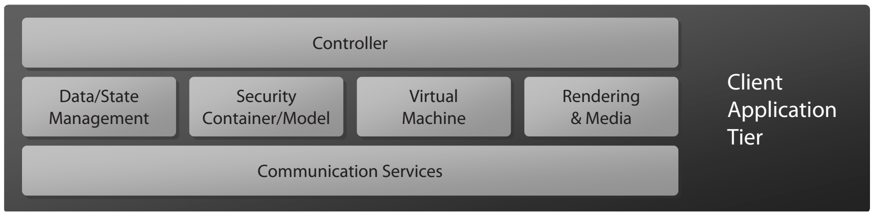

Figure 3.3 – client application architecture

As depicted in Figure 3.3, clients must have far more robust ***communications services*** than a decade ago. In fact, any communication standards, protocols and technologies (such as SOAP[ix], ActionScript Messaging Format, or XML-RPC) have to be implemented on both sides to facilitate proper communications. Client side communications buses also need to monitor the state of communications including potentially both synchronous and asynchronous exchange patterns.

The main ***controller*** of each client application must be capable of launching various runtime environments. This is typically done via launching one or more ***virtual machines*** that can interpret scripting languages or consume bytecode as in Adobe Flash. The architecture for these virtual machines varies greatly depending upon the language used. Some compile an intermediate level bytecode just in time to run a program while others must be launched and make

viii. Representational State Transfer (REST) is an important component of Roy Fielding's Dissertation Architectural Styles and the Design of Network-based Software Architectures - http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
ix. The Simple Object Access Protocol is a W3C Recommendation - http://www.w3.org/TR/soap/

multiple passes over a script (usually once to check it for errors, another time to run the script, and a concurrent iteration to collect garbage and free up memory as it becomes possible to reallocate.

Most modern clients have some form of data persistence and state management. This usually works in conjunction with the clients' communications services to allow the controller to use cached resources rather than attempting to synchronize states if communications are down. Additionally, rendering and media functionality specific to one or more languages is used to ensure the view of the application is built in accordance with the intentions of the application developer.

The security models used by different clients also vary somewhat. The usual tenets are to prevent unauthorized and undetected manipulation of local resources. In distributed computing architectures, identity (knowing who and what) is a major problem that requires a complex architecture to address. Each client side application must be architected in accordance with the acceptable level of risk based on the user requirements.

3.3 Architectural Conventions spanning multiple tiers

While examining the client and service tiers of the reference architecture, developers will note some commonalities. Architects need to employ common models for determining what constitutes an object, what constitutes an event, how an event gets noticed or captured, what constitutes a change in state, and more. As a result, architecture must take note of several common architectural models over all tiers of modern SOAs.

First and foremost, the core axioms of service oriented architecture should be observed. Services themselves should be treated as subservient to the higher level system or systems that use them. If you are deploying services to be part of an automated process management system, the services themselves should not know (or care) what they are being used for.

Services that are designed otherwise are architecturally inelegant for a number of reasons.

First, if services were required to know the state of the overall process, state misalignment would likely result if two services had differing states for even a fraction of a second. In such instances, errors might be thrown when this is detected or worse, developers would have to rely on using a series of synchronous calls to services rather than forking a process into asynchronous calls. As depicted in Figure 3.4, services should remain agnostic to what they are used for. The state of a process or other application using services should be kept within the higher layer of logic that uses consumers to invoke the services.

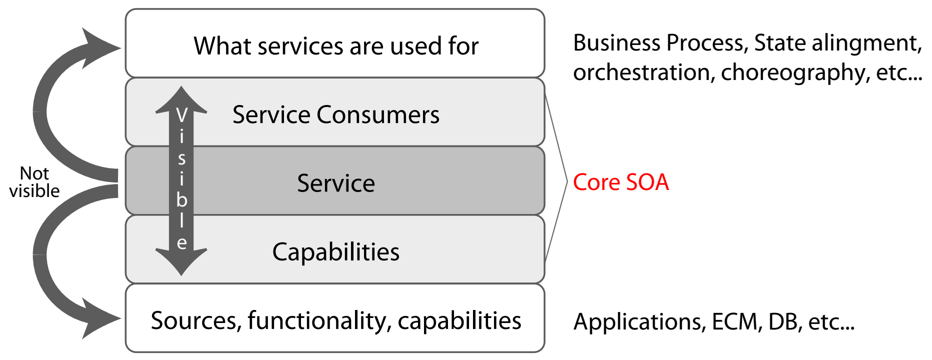
What services are used for
Service Consumers
Service
Capabilities
Sources, functionality, capabilities
Visible
Not visible
Business Process, State alingment, orchestration, choreography, etc...
Core SOA
Applications, ECM, DB, etc...

Figure 3.4 – services within overall architecture

Second, if the overall process stalled or failed for some reason, each service used would have to be notified and rolled back to a previous state. Having services maintain or store the overall state of a process that uses more than one service is an anti-pattern of SOA and should be avoided.

Another core architectural convention is to keep the service consumers agnostic to how the services are delivering their functionality. This results in a clean decoupling of components, another architecturally elegant feature of modern service oriented systems. Having dependencies on knowing the internal working of the services functionality is another anti-pattern of SOA and should also be avoided.

Service composition, the act of building an application out of multiple services, is likewise an anti-pattern of SOA, if composition is defined as per Unified Modeling Language (UML) 2.0[x]. Composition is depicted as a "has a" relationship and the whole is composed of the parts. The correct terminology should be service aggregation. ***Aggregation*** is a "uses a" type of relationship. The differences are quite subtle but nevertheless important to grasp. In composition relationships, the life cycles of parts are tied to the lifecycle of the whole and when the whole no longer exists, the parts no longer exist either. In aggregation, the parts exist independent of the whole and can go on living after the entity that uses them no longer exists. This terminology is common within both OOPSLA[xi] and UML. Regardless, the term "service composition" has been misused widely within the computer industry and will likely prevail as a norm. Architects and developers should pay close attention to the types of binary relationships between components in loosely coupled, distributed systems and bear these definitions in mind.

3.4 Events

Architects and developers using the reference architecture within this paper should also consider the event architecture. Events often must be detected and acted upon. Each specific programming language has a form of event architecture for detection, dispatching messages, and capturing and linking behaviors to events. The main challenge presented in distributed, service oriented systems is that the event model must traverse multiple environments and possibly span multiple domains. Detecting an event in one domain, dispatching a message to a remote system and linking the event to an action in a virtual machine running on the remote system presents multiple challenges. Architects and developers must often bridge disparate systems. Having a common model used by all systems makes the traversal of systems much easier for developers and architects alike.

3.5 Objects

In much the same way they treat events, each disparate environment in a distributed service oriented environment might have a distinct notion of what constitutes an object. Relying on programming environments and languages that are aligned conceptually with respect to objects (that is, "object-oriented") makes the work of architects and developers much easier. Languages such as JavaScript (specifically JavaScript Object Notation or JSON[xii]), Java, ActionScript, and others have alignment on object concepts. *(Note: ECMA's ActionScript 3.0 is much more object-oriented than previous incarnations and is strongly tied to Java).* When a developer must implement a pattern where an object's state must be tracked in a remote location and action taken upon a state change on the object, a common model for object and encapsulation is important.

3.6 Architectural Patterns

As noted in the reference architecture in Figure 3.1, ***architecture and design patterns*** are an important aspect of any architecture.

Patterns are recurring solutions to recurring problems. A pattern is composed of a problem, the context in which the problem occurs, and the solution to resolve this problem. The focus of a documented software architecture pattern is to illustrate a model to capture the structural organization of a system, relate that to its requirements and highlight the key relationships between entities within the system.

x. Unified Modeling Language is owned by the Object Management Group (OMG) and Described here - http://www.omg.org/technology/documents/formal/uml.htm
xi. OOPSLA is an annual conference around Object Oriented Programming, Systems Languages and Applications - http://www.oopsla.org
xii. JavaScript Object Notation (JSON) is RFC 4627 available at http://tools.ietf.org/html/rfc4627

The modern day concept of patterns evolved from work by Christopher Alexander, the primary author of a book called "A Pattern Language"[xiii] which had a great influence on object-oriented programming. The basic concept of the book was a realization that patterns are the same when architecting a city, a block, a house and a room. Each of these entities employs similar patterns.

The concepts of patterns in software architecture have been widely adopted since being modified by the infamous Gang of Four[xiv] and are now an accepted part of the engineering trade.

Patterns can be classified into three broad categories:

Architecture patterns

Architecture patterns are high level patterns on how systems are laid out and how large systems are divided. These typically account for the major components, their externally visible properties, the major functionality of each component, and the relationships between them.

Design patterns

Design patterns provide a scheme for refining the subsystems or components of a software system, and the relationships between them. They describe a commonly recurring structure of communicating components that solves a general design problem within a specific context.[i]

Idioms

Idioms are the lowest-level patterns and may be specific to a programming language. An idiom guides the implementation aspects of components and the relationships between them, using features specific to a given language or environment.

i. Bradner, S. "Key words for use in RFCs to Indicate Requirement Levels." IETF RFC 2119, March 1997; http://www.ietf.org/rfc/rfc2119.txt

4.0 Data and Message Exchange Patterns for Enterprise SOA

The most basic message exchange pattern is a common Request-Response where the parties can simply communicate with each other. This is the basic building block of most SOA interactions and is depicted below.

4.1 Request-Response

Request-Response is a pattern in which the service consumer uses configured client software to issue an invocation request to a service provided by the service provider. The request results in an optional response, as shown in Figure 4-1.

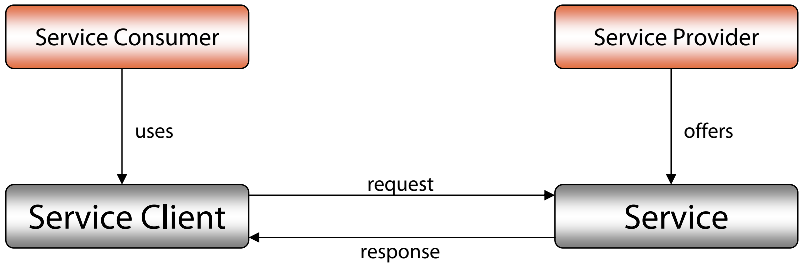

Figure 4-1. SOA Request-Response pattern

4.2 Request-Response via Service Registry (or Directory)

An optional service registry can be used within the architecture to help the client automatically configure certain aspects of its service client. The service provider pushes changes regarding the service's details to the registry to which the consumer has subscribed. When the changes are made, the service consumer is notified of these changes and can configure its service client to talk to the service. This is represented conceptually in Figure 4-2.

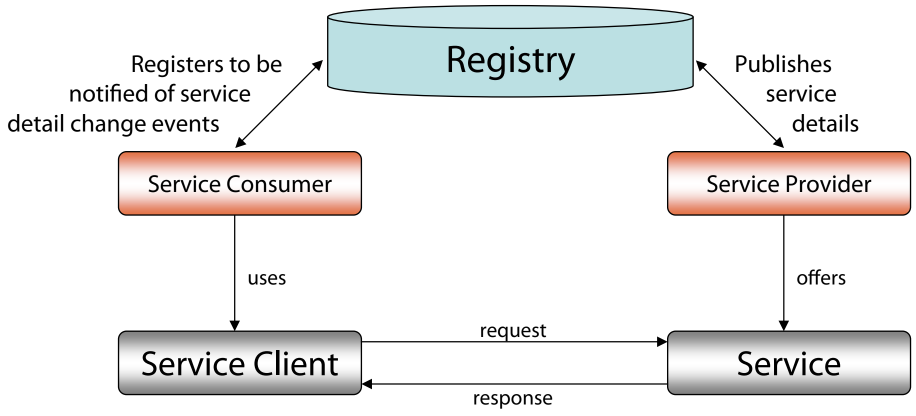

Figure 4-2. SOA Request-Response pattern with a service registry

xiii. http://www.amazon.com/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199

xiv. See "Design Patterns: Element of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

4.3 Subscribe-Push

A third pattern for interaction is called Subscribe-Push, shown in Figure 4-3. In this pattern, one or more clients register subscriptions with a service to receive messages based on some criteria. Regardless of the criteria, the externally visible pattern remains the same.

Subscriptions may remain in effect over long periods before being canceled or revoked. A subscription may, in some cases, also register another service endpoint to receive notifications. For example, an emergency management system may notify all fire stations in the event of a major earthquake using a common language such as the OASIS[xv] Common Alerting Protocol (CAP)[xvi].

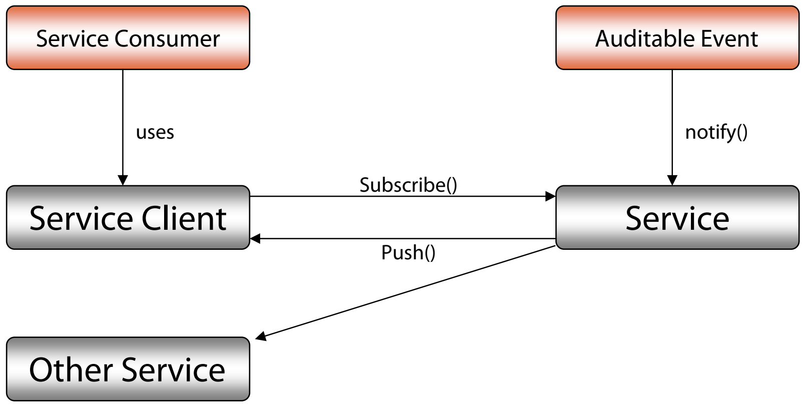

Figure 4-3. SOA Subscribe-Push pattern

Note that this pattern can be triggered by a multitude of events. In figure 4-3, an auditable event is triggering a message being sent to a subscribed client. The trigger could be a service consumer's action, a timeout action, or a number of other actions that are not listed in the example above. Each of these represents a specialization of the Subscribe-Push pattern.

4.4 Probe and Match

A pattern used for discovery of services is the Probe and Match pattern. In this variation, shown in Figure 4-4, a single client may multicast or broadcast a message to several endpoints on a single fabric, prompting them to respond based on certain criteria. For example, this pattern may be used to determine whether large numbers of servers on a server farm are capable of handling more traffic by checking if they are scaled at less than 50% capacity. This variation of the SOA message exchange pattern may also be used to locate specific services. There are caveats with using such a pattern, as it may become bandwidth-intensive if used often. Utilizing a registry or another centralized metadata facility may be a better option because the registry interaction does not require sending the `probe()` messages to all endpoints to find one. By convention, they allow the query to locate the endpoint using a filter query or other search algorithm.

xv. The Organization for the Advancement of Structured Information Systems (OASIS) at http://www.oasis-open.org
xvi. OASIS CAP is a product of the OASIS Emergency Services Technical Committee - http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=emergency

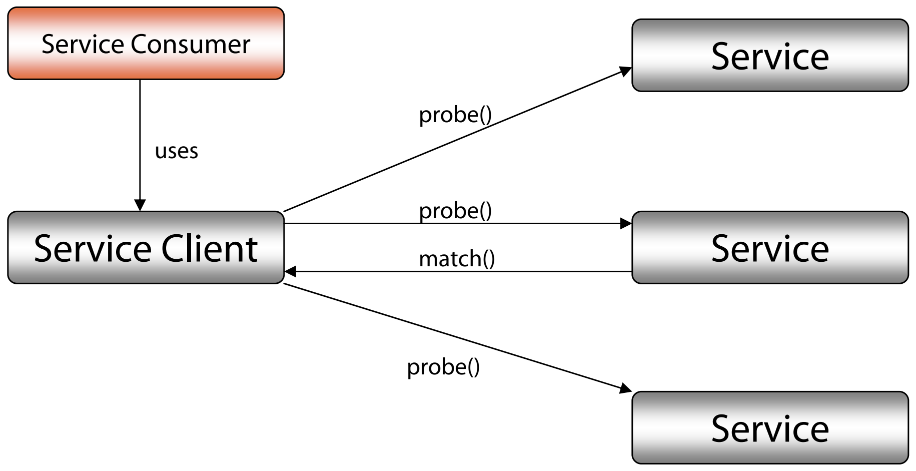

Figure 4-4. SOA Probe and Match pattern

In the Probe and Match scenario in Figure 4-4, the service client probes three services, yet only the middle one returns an associated `match()` message. A hybrid approach could use the best of both the registry and the probe and match models for locating service endpoints. In the future, registry software could implement a probe interface to allow service location without requiring wire transactions going to all endpoints and the searching mechanism could probe multiple registries at the same time.

4.5 Patterns for RIAs

Creating Rich Internet Applications (RIAs) requires a level of data management that goes beyond the traditional Request-Response model. Providing a richer, more expressive experience often requires more data-intensive interaction and introduces new challenges in managing data between the client and server tiers.

Data synchronization is a key concept and requires states to be shared among multiple machines. These are usually the clients who have subscribed to the state of an object somewhere within the tier of a distributed system as depicted in Figure 4.5.

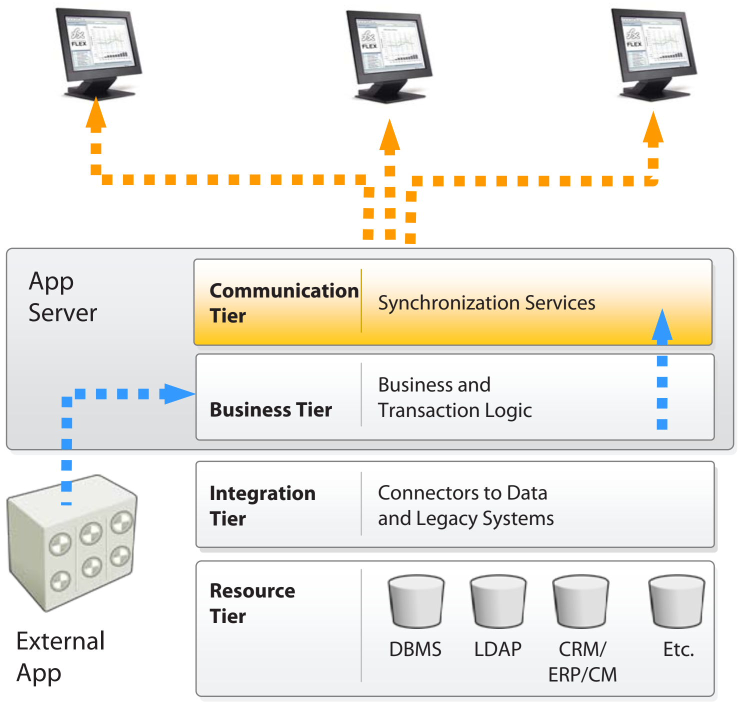

Figure 4.5 – data synchronization across multiple clients (courtesy James Ward).

4.6 Data paging

Some services automatically facilitate the paging of large data sets, enabling developers to focus on core application business logic instead of worrying about basic data management infrastructure.

Modern service oriented clients and server infrastructures automatically handle temporary disconnects, ensuring reliable delivery of data to and from the client application.

4.7 Data push

Some services offer data-push capability, enabling data to automatically be pushed to the client application without polling (contrast this pattern to the "Subscribe-Push pattern listed above). This can be done via intuitive or inference methods to ensure data is provided as required. This highly scalable capability can push data to thousands of concurrent users, providing up-to-the-second views of critical data, such as stock trader applications, live resource monitoring, shop floor automation, and more.

Data push can be further specialized into broadcast, unicast, multicast, and several other specializations of the basic pattern.

5.0 A Final Word

This white paper has been prepared to share ideas about data interaction patterns within SOA and to illustrate some common concepts with a service oriented environment. It is based on input provided by a number of people from different companies and is not considered the work of any one company. It is free to share, use, quote, and post wherever and however you want.

Service Oriented Architecture will likely remain the mainstay of technology platforms for the foreseeable future. It is our hope that the companies who have contributed to this will continue to write more on specialized patterns of SOA.

About the Authors

Duane Nickull is a Senior Technical Editor at Adobe Systems – http://technoracle.blogspot.com

Laurel Reitman is a Senior Architect with Adobe Systems

James Ward is a Technical Evangelist for Flex at Adobe and Adobe's JCP representative to JSR 286, 299, and 301 – http://jamesward.org

Jack Wilber is a freelance writer and developer. He draws on more than ten years of experience in software development and holds a B.S. in computer and electrical engineering from Carnegie Mellon – http://www.jgwconsulting.com

Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704
USA
www.adobe.com