A Comparison of Service Oriented Architecture with other Advances in Software Architectures

Benjamin Kanagwa and Ezra K. Mugisa

Service Oriented Architecture (SOA) allows software systems to possess desirable architecture attributes such as loose coupling, platform independence, inter-operability, reusability, agility and so on. Despite its wide adoption in the form of Web services, many stakeholders both from academia and industry have limited understanding of its underlying principles. This has led to faulty implementations of SOA and in some cases, it has been implemented in places where it is not suitable. In this paper, we investigate and show the relationship between SOA and other advances in software architecture. The paper relates SOA to Architecture Patterns, Components and Connectors. We advance the view that SOA’s uniqueness and strength does not lie in its computational elements but in the way it enables and handles their interaction. In this way, we facilitate the understanding of SOA in terms of other advances in software architectures that are already well understood. We believe that a good understanding of SOAs in terms of other advances in software architectures will help to reap its enormous benefits.

Introduction

SOA has gained more popularity of recent, largely due to web services (Hashimi, 2004) and partially due to natural evolution in software engineering (Tsai, 2005). At the same time, there is a systematic progression from object oriented systems, distributed objects, components and then service oriented systems. This evolution in software engineering shows a systematic move from tightly coupled and platform specific systems to more robust loosely coupled platform-independent systems.

Despite this progression, and widespread interest in SOAs, most publications on the subject fail to explain SOA or simply assume it merely defines a synonym for a stack of extensible markup language (XML) web service protocols and standards (Stal, 2006). Stal argues that developers need to understand the service oriented paradigm from the architecture perspective in order to leverage SOA implementations. This lack of understanding has already manifested itself in faulty implementation of SOA as highlighted by Loughran and Smith (2005). We argue that these faulty implementations that carry remote procedure call sentiments can only be addressed by understanding the core principles of SOA. Since we cannot divorce services from
SOA, we start by addressing the problem from the core. The focus is to provide an understanding of SOA in terms of well established concepts in software architectures.

It has been reported in literature (OASIS, 2006) that SOA provides a way of building architectures and builds on existing advances in software architecture (Szyperski, 2003), (Gokhale et al., 2002). We view SOA as an alternative approach to building architecture of systems. The architecture solution space includes well formulated approaches such as: Component Based Architectures (CBA), Connectors, Architecture Patterns/Styles and Architecture Description Languages (ADLs). Given that these concepts are well developed and understood, it is important to show how they relate to SOA. We are particularly interested in those advances that are in line with providing solutions to the software architecture problems. As part of our investigation we show the specific features that have been enhanced by SOA to make it distinct from its predecessors.

Our definition for SOA and service is from our previous work (Kanagwa and Mugisa, 2007) where we define a system (application) that is considered to have a SOA. From this definition, we can establish the principles built within SOA systems. A service is defined as a set of functionality that exposes a resource for use by others and is invoked directly under the conditions it specifies. We say that an application is a SOA system if it is partitioned into sets of independent functionality that interact directly, independent of context.

The concepts in the above two definitions are based on the definitions provided by (Baker and Dobson, 2005) and (OASIS, 2006). In (Baker and Dobson, 2005), a service is defined as “a set of functionality provided by one entity for the use of others”. We feel that it is important to emphasize direct interaction to avoid confusion with technologies such as Common Gateway Interface (CGI) (accessed through web servers), and Components, that need component frameworks through which they interact.

**SOA Context**

We take a brief look at the use of the term SOA. It has been described as a framework, as a paradigm (OASIS, 2006) as a collection of services, plus several other definitions. What is clear from these definitions is that they do not represent the same thing. At this point, we would like to distinguish the two uses of the term SOA. In some contexts, SOA is used to refer to an abstract architecture (architectures that do not represent concrete systems). However, SOA has another meaning to it in which it is used to refer to concrete systems (running applications). The later is common from the industry practitioners.

In the first context, SOA does not represent architecture of a specific system just like Common Request Broker Architecture (CORBA). Definitions such the framework, paradigm refer to this dimension of SOA. Definitions such as collection of services are inline with the second dimension of SOA. In the second context, the definition aims at answering the question, when is a system service
oriented? The ideal position should be that systems designed as per the first SOA context imply the second context. In other words, if SOA is a framework or paradigm, systems designed as per this framework or paradigm should lead to systems whose architecture is service oriented. Unfortunately this is not the case.

As a framework, paradigm or way of building architectures, SOA involves what we have described under the section “architecture patterns for SOA”. In addition SOA specifies how to specify and describe services, how to facilitate the actual interaction between services. Although how to ‘publish’ and 'find' services are key principles in the SOA framework, they are not part of architectures of resulting systems.

Just like component repositories do not contribute to the software architecture, the way the services are published and discovered is not really part of running SOA.

Contribution

Our major contribution is tracing the evolution of SOA and expressing SOA in terms of other advances in software architectures that are already well established and understood. In this way, we facilitate understanding of SOA and foster its proper usage. Proper understanding and usage of SOA will leverage its benefits.

The rest of the paper is organised as follows. We discuss related work, investigate the evolution of SOA, show how SOA relates to other advances in software architectures, show features inherited by SOA from its predecessors and provide a conclusion.

Related Work

A comparison of Service-Oriented and Distributed Object Architectures is given in (Baker and Dobson, 2005). They state that: “while superficially similar, the two approaches exhibit a number of subtle differences that, taken together, lead to significant differences in terms of their large-scale software engineering properties such as the granularity of service, ease of composition and differentiation properties that have a significant impact on the design and evolution of enterprise-scale systems”.

Stal (2006) suggests the use of architecture patterns and blueprints to explain the architecture principles of SOA. The work is based on the driving forces to define a model for the service oriented context. The work addresses the architecture patterns necessary to address the driving forces. However, the work does not do much to facilitate the understanding of SOA. The work focusses on implementation and design patterns. Our effort is aimed at explaining SOAs by stressing the relation of SOA with existing architecture concepts (patterns inclusive).

Evolution of SOA

We have noted earlier that SOA does not represent a concrete system but provides a way of building architectures. So it is part of the software architecture solution
space. We describe the emergence of SOA using figure 1. The aim is to put SOA in context based on its drivers, architecture properties, architecture realisation and instances.

**Fig 1: Showing relation between SOA concerns (drivers), architecture features and architecture**

In figure 1, we argue that architecture evolution is precipitated by limitations of the architecture approaches in addressing existing and emerging situations. We call the limitations architecture drivers. In other words, these limitations create the need for a new architecture approach. The architecture drivers point to specific architecture properties. The architecture properties translate into an architecture framework or style. Finally the realization of the software architecture is obtained. The architecture realization must satisfy the drivers and this completes the relationship.

**Architecture Drivers for SOA**

SOA is a distributed computing architecture (OASIS, 2006) and so are the predecessors. Remote Procedure Call (RPC), CORBA, Distributed Common Object Model (DCOM), and Remote Method Invocation (RMI) are (were) the most widely used and well known distributed computing architectures. They were developed at a time the Internet was not widely spread or discovered to be a potential distributed computing platform. They therefore failed to rise to the challenges of the Internet as a global computing platform. To fully reap the benefits of the internet, a new way of building software architectures was needed. SOA carries elements of existing architecture approaches but emphasizes the interaction of its architecture elements.

Baker and Dobson (2005) argue that SOA differs from Distributed Object Architectures in a number of aspects that lead to combined significant differences in terms of large scale engineering properties. In (Alencar et al., 2002), we learn that current software applications can be separated into a basic concern and a number
of special purpose concerns. The basic concern is represented by the fundamental computational algorithms that provide the essential functionality relevant to an application domain and the special purpose concerns relate to other software issues, such as user interface presentation, control, timing, and distribution. SOA is driven by the quest to address the special purpose concerns that relate to large scale systems. These concerns are mainly interaction issues. SOA’s strength does not lie in its computational elements but in the way it enables and handles their interaction. Interaction manages the behavior of elements and therefore has substantial influence on the overall system. By focusing on interaction, SOA is able to mitigate the limitations of its predecessors.

**Soa And Software Architectures**

Most work in SOA emphasizes enabling technologies such as in web services. We assess the contribution of SOA to the software architecture solution space as an alternative solution or enhancement of other advances in software engineering.

**Architecture Patterns and SOA**

SOA is modeled as a relationship of three kinds of participants (roles): the service consumer (client), the service registry, and the *service provider*. This view of three participants is shared by (Papazoglou, 2003), (Hashimi, 2004), (Pilioura and Tsalgatidou, 2001), (Talaei-khoeil et al., 2005), (Huhns and Singh, 2005) and (Gottschalk et al., 2002) among others.

At the center of SOA are the above three roles. It starts with the service provider publishing its existence in some registry, followed by the service consumer finding the service provider by querying the service registry. The service consumer then binds directly to the service provider. This translates to the publish–find–bind architecture pattern. The realization of this pattern is left to different instances of SOA. In (Hull et al., 2003, Wu and Chang, 2005), they relate the SOA pattern with other advances in software architecture especially broker, peer-to-peer and client-server architecture patterns. Web services is the most developed instance of SOA, and we draw valuable lessons from its implementation. Table shows a summary of the interaction styles from web services that are closely related to SOA. The styles in the table are not definitive to SOA because other possibilities are possible.

It is desirable that services be published to a registry. The publish interaction employs the client-server interaction style that is characterized by request–response. However, SOA imposes no restriction.
Any form of publishing can be used. For example, if the target consumers are known, the service descriptions can be directly sent to the consumers. This is very likely in highly specialized services such as in the military where the consumers are restricted.

Find relates to discovering binding information for a service. Typically, it is the connection between the consumer and registry. The pattern of finding a service can be carried out using client-server interaction style. The request is inform of a ‘service specification’ and the reply is the ‘service description’. In this case, the consumer is the client and the registry is the server. Other possibilities exist. Direct interaction between the consumer and provider (example of small in-house developers) or in specialized services discussed above. It is also possible for a consumer to make a provider-by-provider search until it finds the right service. Although many possibilities for finding services are possible (especially if fewer services are involved), automating service interactions requires client-server interaction.

Bind, involves the actual interaction between the service provider and consumer or simply services. A service has multiple operations and each interaction is independent of others. In respect to a single interaction, one service acts like a client and the other like a server leading to client-server interaction style. However, the reply is optional and the services are not really clients and servers as defined in client-server architectures .The services simply assume the roles when required for a specific interaction.

Whereas publish and find are important aspects of SOA, they are not part of the resulting architecture. The resulting architecture is the collection of services and how they interact with no regard on the ‘publish’ and ‘find’ styles. The publish-find styles enable quick and flexible binding, but do not manage the service binding or provide rational for choice of the services. They are therefore not part of the architecture. The architecture of a service oriented is defined once the service have been found and bound together. Architecture reconfiguration is possible through dynamic discovery and binding.

Thus, the focus of SOA systems should be on enabling quick and flexible binding between services. SOA provides the underlying framework for interaction and dynamic behaviour of services. We believe that this service interaction is reusable across all service oriented systems and is what characterizes SOA.
Architecture Connectors and SOA

Interactions in software architectures are modeled by connectors (Shaw and Garlan, 1996). It has been argued that even components that do not use connectors explicitly have composition operators that can be identified as connectors at different levels of abstraction (Lau et al., 2005). We note that this is true for all systems that are partitioned into subsystems. Such subsystems must communicate to be able to work as a unit. Connectors therefore exist in all systems but in different forms and levels of abstraction. The abstraction levels range from low level programming to more explicit connectors at the architecture.

Even though connectors are not explicitly mentioned in relation to SOA’s, it is easier to appreciate the strength of interactions in software architecture by drawing comparisons with connectors. SOA’s main uniqueness is derived from its interaction mechanisms that control the overall system rather than structure elements. In relation to connectors, the significance of interactions is given by (Medvidovic et al., 2000), (Bˇ alek and Plˇ ašíl, 2001) and (Guo et al., 2005) among others.

Whereas the main system functional blocks are components, the properties of the system also strongly depend on the character of the component interactions (Bˇ alek and Plˇ ašíl, 2001). They identify benefits of connectors as: increased re-usability, direct support for distribution, location transparency, mobility of components in a system, support for dynamic changes in the system’s connectivity and so on. Medvidovic et al. (2000), point out that in large, and especially distributed systems, connectors become key determinants of system properties, such as performance, resource utilization, global rates of flow and security.

Connectors help to localize information about interaction of components in a system such that information is no longer spread over all communicating components (Guo et al., 2005). Therefore, connectors have significant impact on the quality attributes and offer a basis for abstracting interactions and reasoning about systems at the architecture level. We note that typical service based systems operate by interacting among constituent services.

SOA focuses on the special purpose concerns that relate to inter-operability and loose coupling. It borrows some of the concerns from its predecessors. However, SOA refines the concerns to provide significant impact on the systems that use it.

Connectors can exist in explicit or implicit form. Although SOA does not define explicit connectors, it is clear that its implicit connectors shape the characteristics of SOA. SOA does not define connectors explicitly but defines the infrastructure for constructing the connectors. The infrastructure generally includes the interface specification, descriptions, message formatting and messaging protocols.

Predecessors of SOA

We would like to think that SOA builds on existing knowledge, particularly architecture approaches. The question to answer here is which old system does SOA compare with or extend? Or what does a service extend? To put the question
in another way, which architecture features does it build on? This question has been answered by (Gokhale et al., 2002) and (Szyperski, 2003). Gokhale et al. (2002) makes a comparison between Web services and CORBA. Szyperski (2003) suggests that some elements of component technology apply to services.

What (Szyperski, 2003) and (Gokhale et al., 2002) do not answer however, is what exactly do we carry forward to SOA? (Wang and Fung, 2004) provide an interesting comparison of Object Oriented Architecture (OOA), Component Based Architecture (CBA) and Service Based Architecture (SBA). It is clear from the characteristics -that SOA emphasizes interaction rather than structure. For example, a one-to-one comparison structural comparison of Web services and CORBA (Gokhale et al., 2002) does not yield much. It only reveals philosophical differences that translate to efficient interaction and only contribute to the overall architecture of the resulting systems.

Features from Predecessors

From its main predecessors:-Distributed Component Model (DCM), a subset of CBSE, SOA carries the following features: composability, separation of concerns, loose coupling and abstraction. SOA combines benefits and challenges from COTS and CBD except that services are consumed from wherever they are. We note that SOA relaxes the restrictions on the features to make it unique from its predecessors.

Composability

From CBSE, we carry forward the belief that complex systems can be built from smaller elements. Unlike CBSE, SOA is not constrained by a specific component model. Component models are in fact the major limitation to interoperability with existing component models such as DCOM, CORBA, EJB not able to easily interoperate. The importance of component models in DCM and CBSE is highlighted by (Lau and Wang, 2005) and (Baker and Dobson, 2005). Comparing SOA and component models, SOA is a relaxation of the restrictions on component based systems. There is no requirement for a specific internal structure, no component model semantics, no containers and application servers. This relaxation fosters software agility that requires software to change frequently and gracefully.

Separation of concerns

This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability and reusability (Aksit et al., 2001). In SOA the major concerns are integration of autonomous systems. Integration becomes complex if the systems in question do not match in terms of protocols used and data formats. Thus, the critical starting point addressed by SOA is the way systems are exposed and consumed. SOA requires that systems are accessed through platform independent interfaces.
However, a key concept in Object Oriented Programming (OOP) that is not necessary in SOA is inheritance mainly because SOA is not a programming model and services are self-contained, independent entities whose internal details are invisible. Services do not need to inherit any functionality since they can include the functionality through standardised service interactions.

**Loose coupling**

Loose coupling is the most pronounced architecture property associated with SOA. Loose coupling is not really a new concept that is unique to SOA. It exists in CBSE, except SOA is designed to be more loosely coupled. SOA achieves more loosely coupling through its emphasis on platform neutral interfaces, platform neutral descriptions, message based interactions and self-contained services. In tightly coupled systems, architecture elements are designed for each other, while in loosely coupled systems, architecture systems are designed to interoperate.

**Abstraction**

SOA extends the concept of abstraction and encapsulation. Unlike in OO which hides internal details (data and methods), SOA aims at hiding all causes of integration problems. Stal (2002) lists the causes of heterogeneity as “network technologies, devices, and OSs; middleware solutions and communication paradigms; programming languages; services and interface technologies; domains and architectures; and data and document formats.”

SOA encapsulates a specific set of discrepancies in its domain of application. By hiding these differences, services within the service oriented architecture can be accessed seamlessly. The differences are typically hidden by using a common set of standards. The differences may be in terms of data structures, communication protocols or platforms. For example, SOA’s that target the Internet as the mode of delivery, will require standards that hide heterogeneity in platforms, while a simple SOA that is part of an operating system such as accessing devices, will require standards that present the devices in the same way. A typical example in UNIX where everything is a file. In terms of architecture, it implies that standards play a critical role. Through use of standards, protocols and common vocabulary, the differences in participants are resolved thereby allowing all the participants to appear uniform.

Generally, SOA does not eliminate the heterogeneity, but simply hide the heterogeneity causes. The systems and technologies remain heterogeneous (Stal, 2002) but their interfaces and collaborations are standardized. This is a core concept of SOA and the choice of interface specification and standards is left to different instances of SOA. The choice

and design of the standards must be carried out with care lest they will be a bottleneck. They must be simple to use and comprehend without compromising the efficiency. Simple standards tend to be very verbose while short standards are normally hard to understand. A key to this is a compromise between the two
extremes. According to Almeida et al. (2003), the protocols and standards that enable technology abstraction should be suitable and intuitive for application developers that develop and maintain applications in different technology domains.

Uniform access as a requirement for as a strength for SOA is noted by (Curbera et al., 2001), (Baker and Dobson, 2005). Curbera et al. (2001) “a key goal of WS framework is to produce a common representation of applications which use diverse communication protocols and interaction models while at the same time enabling Web services to take advantage of more efficient protocols when they are available at both ends of the interaction”.

**Conclusion**

We have showed the relation between SOA and other advances in software architectures. In so doing, we have clarified most architecture issues surrounding SOA. We have also showed some of the exact features carried by SOA’s from its predecessors. Through comparison of SOA with other advances in software architecture, we have advanced the view that SOA’s uniqueness and strength does not lie in its computational elements but lies in the way it enables and handles their interaction. Whereas SOA relaxes several features from its predecessors, we have showed how such relaxations impact on the architecture of the resulting system. We have backed this claim by looking at the contribution of interactions (connectors) to software architectures.

Therefore, we recommend that future research in SOA should focus on optimizing that view.

**References**


