Challenges and Opportunities for Formal Specifications in Service Oriented Architectures

Gustavo Alonso

Systems Group
Department of Computer Science
ETH Zurich, Switzerland
{alonso}@inf.ethz.ch
http://www.inf.ethz.ch/~alonso

Abstract. Service Oriented Architectures (SOA) are currently attracting a lot of attention in industry as the latest conceptual tool for managing large enterprise computing infrastructures. SOA is interesting from a research perspective for a variety of reasons. From the software engineering side, because it shifts the focus away from conventional programming to application integration, thereby challenging many of the premises around improving development by improving programming languages. From the middleware point of view, SOA emphasizes asynchronous interaction, away from the RPC/RMI model, and thus brings to the fore many of the inadequacies of existing software and hardware platforms. From the formal specification perspective, however, SOA offers many opportunities as one of the key ideas behind SOA is the notion of capturing the interactions at a high level and letting the underlying infrastructure take care of the implementation details. For instance, the emphasis in SOA is the main reason why workflow and business process technologies are experiencing a new revival, as they are seen as a way to formally specify complex interaction patterns. This presentation covers the main ideas behind SOA and why they are an excellent basis to provide a more formal basis for the development and evolution of complex systems.

Keywords: Multi-tier architectures, Service Oriented Architectures, Web Services, Workflow, Business Processes, Declarative Languages.

1 Introduction

Service Oriented Architectures (SOA) are the latest approach to deliver a better understanding and improved techniques to master the complexities of the modern enterprise architectures. SOA differs form past attempts in several fundamental ways. First, it is language independent and makes no assumption about the underlying programming model. Second, communication is no longer based almost exclusively on request-response patterns (RPC/RMI) but the emphasis is on asynchronous events and messages. Third, SOA sees the development of new applications and services mainly as the integration and composition of large

K.M. van Hee and R. Valk (Eds.): PETRI NETS 2008, LNCS 5062, pp. 1–6, 2008.

scale services and applications rather than as a smaller scale programming problem. These differences arise from the lessons learned in the last two decades and represent a significant step forward. These differences also create very interesting opportunities and pose a wide range of challenges that will be discussed in this presentation.

2 Background

2.1 Multi-tier Architectures

There is a general classification commonly used for multi-tier systems that is useful for discussing enterprise architectures [ACKM03]. It is based on a number of layers, each one build on top of the lower one and providing distinct functionality. The layers are logical entities and can be implemented in a variety of ways, not necessarily always as separate entities. The lowest layer I the classification is the Resource Manager layer, which provides all the data services necessary to implement the upper layers. The resource manager is often implemented as a database but it can also be simpler (e.g., a file system) or much more complex (e.g., an entire sub-systems also implemented as a combination of layers). The Application Logic is built directly on top and it contains the code that implements the functionality that defines the application. The application logic layer can be as simple as a monolithic program or as complex as a fully distributed application. The final layer is the Presentation Layer, or the part of the system that deal with external interactions, regardless of whether they are with users or with other applications. The presentation layer should not be confused with a client, which is a particular form of distributing the presentation layer. The presentation layer is not the final interface (e.g., a web browser) but the functionality that prepares the information to be sent outside the application (e.g., the web server). The properties of an enterprise architecture can be studied based on the integration patterns that arise as the different layers are combined and distributed in different forms. There are four basic patterns: one tier, two-tier, three tier, and multi-tier architectures.

One tier architectures arose historically from mainframes, where the applications where implemented in a monolithic manner without any true distinction between the layers. Such design can be highly optimized to suit the underlying hardware; the lack of context switches and network communication also removes a great deal of overhead compared with other architectures. An important characteristic of one tier architectures is that the presentation layer is part of the architecture and the clients are passive in the sense that they only display the information prepared before hand in the one tier architecture. Decades ago this was the way dumb-terminals operated. Today, this is the way browser based applications based on HTML work and the basis for the so called Software as a Service (SaaS) approaches, where the client does not run any significant part of the application. Although SaaS is not implemented as a one tier architecture on

the server side, the properties it exhibits from the outside are similar to those of a one tier architecture.

Two tier architectures take advantage of distribution by moving the presentation layer to the so called client while leaving the application logic and the resource manager in the so called server. The result is a system with two tier that are generally tightly coupled as the client is developed for a particular server, since the client is actually the presentation layer of the global application. Two tier architectures represented historically a very important departure in terms of technology as they appeared at the same time as RPC and led to the definition of many different standard interfaces. Yet, they face important practical limitations: if a client needs to be connected to different servers, the client becomes the point of integration. From an architectural point of view, this is not scalable beyond a few servers and creates substantial problems maintaining the software. On the server side, coping with an increasing number of clients is complicated if the connections are stateful and the server monolithic as it then becomes bottleneck.

In part to accommodate the drawbacks of client server architectures, and in part to provide the necessary flexibility to adapt to the Internet, two tier architectures eventually evolved into three tier architectures. The notion of middleware, as the part of the infrastructure that resides between the client and the server, characterizes three tier architectures quite well. The middleware provides a single interface to all the clients and to all the servers; it serves as the intermediate representation that removes the need for application specific representations; and becomes as supporting infrastructure to implement useful functionality that simplifies the development of the application logic. In its full generality, the middleware helps to connect a fully distributed infrastructure where not only every tier is distributed. Each tier can in turn be distributed itself as it happens today in large scale web server systems.

From three tier architectures, a variety of integration patterns have evolved in the last two decades that range from RPC based infrastructures (e.g., CORBA) to message oriented middleware (MOM) where the interactions are asynchronous and implemented through message exchanges rather than request/response operations. MOM was for a while a useful extension to conventional (RPC based) middleware. Slowly, however, developers and architects started to realize the advantages of message based interaction over the tight coupling induced by RPC systems. Today asynchronous interactions are a very important part of the overall architecture and, to a very large extent, have become a whole integration layer by themselves.

From an architectural point of view, few real systems match one of those architectural patterns exactly. In particular, complex systems are often referred to a N-tier architectures to reflect the fact that they are a complex combination of layered systems, distributed infrastructures, and legacy systems tied together often with several layers of integration infrastructure. These complex architectures are the target of this presentation as they form the basis of what is called an enterprise architecture.

2.2 Web Services and Service Oriented Architecture

Multi-tier architectures integrate a multitude of applications and systems into a more or less seamless IT platform. The best way to construct such systems is still more an art than a science although in the last years some structure has started to appear in the discussion. During the 90's middleware platforms proliferated, with each one of them implementing a given functionality (transactions, messaging, web interactions, persistence, etc.) [A04]. Most of these platforms had a kernel that was functionally identical to all of them, plus extensions that gave the system its individual characteristics. In many cases, these systems were used because of the functionality to all of them (e.g., TP-monitors that were used because they were some of the earliest and better multi-threading platforms). The extensions were also the source of incompatibilities, accentuated by a lack of standardization of the common aspects [S02].

The emphasis today has changed towards standardization and a far more formal treatment of the problems of multi-tier architectures. The notion of "enterprise architecture" is a direct result of these more formal approaches that start distinguishing between aspects such as governance, integration, maintenance, evolution, deployment, development, performance engineering, etc. Two of the concepts that dominate the discourse these days are web services and service oriented architectures. Although in principle they are independent of each other and they are not entirely uncontroversial, they are very useful stepping stones to understand the current state of the art.

Web services appeared for a wide variety of reasons, the most important ones being the need to interact through the Internet and the failure of CORBA to build the so called universal software bus [A04]. Once the hype and the exaggerated critique around web services are removed, there remain a few very useful concepts that are critical to enterprise architectures. The first of them is SOAP, a protocol designed to encapsulate information within an "envelope" so that applications can interact regardless of the programming language they use, the operating system they run on, or their location. SOAP is interesting in that it avoids the pitfalls of previous approaches (e.g., RPC, CORBA, RMI) by not making any assumptions about the sender and the receiver. It only assumes that they can understand one of several intermediate representations (bindings). The second important concept behind web services is WSDL, or the definition of services as the interfaces to complex applications. WSDL describes not only the abstract interface but also the concrete implementations that can be used to contact that service. By doing so it introduces the possibility of having a service that can be used over several protocols and through different representations, the basis for adaptive integration and an autonomic integration infrastructure. Web Services do not need to involve any web based communication. In fact, they are extremely useful in multi-tier architectures as the interfaces and basic communication mechanism across systems and tiers. Dynamic binding (such as that implemented in the Web Services Invocation Framework of Apache) allows to give a system a web service interface and lets the infrastructure choose the best possible binding depending on the nature of the caller. A local piece of code

will use a native protocol, a remote caller using the same language will use a language specific binding, a remote a heterogeneous caller will use an XML/SOAP binding. It is also possible to link request/response interfaces to message based interfaces and vice-versa, a clear indication of the potential of the ideas.

The notion of service used in Web Services [ACKM03] has proven to be very useful, more useful that the notions of components or objects used in the past. Services are large, heterogeneous, and autonomous systems offering a well defined interface for interacting with them. They might or might not be object oriented, and it is irrelevant how they are implemented. The key to access them is to support one of the bindings offered by the service. From this idea, it is only a short step to the notion of Service Oriented Architectures, i.e., enterprise architectures where the different elements are connected through service interfaces and where the interactions happen through the interfaces but preferably using asynchronous exchanges like those used in MOM systems. Thus, SOA is a new attempt at realizing the vision of the universal software bus that CORBA started. This time, however, with a much wider understanding of the problems and forcing the programming language concerns out of the picture. SOA is not about programming, like CORBA was (or is). SOA is about integration and about how to best implement architectures that are based on integrating software systems.

Web Services and SOA have been around for just a few years but they have brought with themselves enough interesting concepts that their ideas will become part of the solution, regardless of what form this solution eventually takes. In many ways, they are the first step towards eliminating "spaghetti integration" and moving towards more structured and modular approaches to application integration. In the same way that it was a long way from the initial structured programming languages to today's modern object oriented languages, SOA and Web Services are the first step towards better tools and a more sophisticated understanding of application integration.

SOA has allowed system architects to formulate many problems in a clearer manner than it was possible before. From the problem of defining services, to the types of interactions between these services, including the organization of services as well as the associated problems of ensuring quality of service from design/development time, SOA has helped developers to define a proper framework to tackle such problems in an orderly manner.

3 High Level Programming in SOA

This presentation argues that SOA provides an ideal starting point for a more formal approach to the design, development, deployment, and maintenance of complex, distributed, enterprise architectures. As mentioned in the abstract, SOA is the reason why workflow management systems and business processes are again in fashion. SOA proposes to work at the level of services: large scale applications hidden behind a service interface that can support asynchronous interactions. SOA is about how to build and connect those services into larger

entities. This is very similar to what business processes aim to do: combine a variety of applications into a coherent process that matches a given set of business requirements. It is not surprising then that SOA and business processes have by now become terms that are often mentioned together.

The presentation will provide a historical overview of how multi-tier architectures have evolved and how their evolution led to SOA and Web services. The discussion will include what are the significant differences between SOA and approaches like CORBA or general middleware. Based on this background (briefly explained as well in this extended abstract), the presentation will then focus on what are the advantages of SOA from a technical point of view as well as on the challenges and opportunities for using formal specifications in the context of SOA.

The opportunities SOA offers are cleaner interfaces, a wide range of technical solutions to solve the integration problem, and a cleaner architecture where integration is the primary goal. The challenges to solve before we can take advantage of these opportunities are the embedding of these ideas into conventional programming languages (or within more modern, more suitable languages), the complexity of asynchronous systems, and the need to set reasonable goals for the formal specifications.

References

[ACKM03] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer, Heidelberg (2003)

[A04] Alonso, G.: Myths around Web Services. In: Bulletin of the Technical Committee on Data Engineering, December 2002, vol. 25(4) (2002)

[S02] Stonebraker, M.: Too much Middleware. Sigmod Record (March 2002)