

Service-Orientation: A Brief Introduction

HAMID BEN MALEK

May 24, 2005

Abstract

This article is a very short introduction to Service-Orientation. It merely scratches the subject. We did not have time to cover more topics or go into more details because of a deadline to release it. It is very likely that this paper will be upgraded in the future to cover more topics with a deeper analysis.

Contents

1	Introduction	1
2	The “No Free Lunch” Principle	1
3	Asynchronous is the force	2
4	Loose Coupling	3
5	Service-Oriented Architecture	5
6	Service Orientation Principles	9
6.1	Explicit Service Boundaries	9
6.2	Service Autonomy	9
6.3	Shared Contracts and Schemas	10
6.4	Policy-based Compatibility	11

1 Introduction

I have heard many myths and inconsistencies about service-orientation. One of these myths is that SOA is a very confusing topic because very few people know what it really is if they ever know. Another inconsistency I heard is something that goes like this “I do not believe in the concept of SOA Fabric. I think SOA could be done in many ways and not necessary use the Fabric concept”.

SOA seems a confusing subject not because it is. The fact that there are very contradicting discourses about SOA on the internet does not mean that the subject is confusing or not known precisely. The inconsistency of the discourse is not specifically related to SOA. This can happen to any other topic. For example, take a graduate level topic in mathematics, and let all undergraduate students take a shot at it. You will get very contradicting and various discourses. If IT architects/engineers have different discourse about SOA, it is not because SOA is not known or that it is confusing. There are two major facts here. The first fact is that SOA is a buzz word, and as such, everybody wants to talk about it. If everybody talks about it, you will certainly get very contradicting discourses, even from people who are supposed to be in a field very close to the subject (like undergraduate math students when asked about some math topic of graduate level). The second major fact is that the few people who actually know exactly what SOA is, do not talk about it and refuse to say more (because of corporate strategy). And there are a third category of people who actually say things about SOA to promote their own technology as being SOA.

Concerning the SOA Fabric concept as not necessary the right approach, this is the silliest idea I ever heard. The SOA Fabric means nothing more than the “messaging environment” used by a service-oriented system. A service-oriented architecture involves a set of services (and consumer applications) integrated together. The “messaging environment” (or communication infrastructure) is what makes these services and applications integrated. There is always a messaging environment (even if it just coincides with the Web). One cannot say “we don’t need a messaging environment” because anything that integrates distant applications and services is a sort of “messaging environment”. The SOA Fabric just happens to mean the “messaging environment as used in SOA systems” and no more. Therefore, it is not possible to say that the SOA Fabric is not necessarily needed in SOA.

2 The “No Free Lunch” Principle

Years ago when I was preparing my PhD thesis in pure mathematics, my advisor, during our mathematical conversations, always used to say the following sentence “There is no free lunch”. It used to irritate me a bit, not because I did not understand what he meant by it, but for a different reason. Mathematicians from around the world, despite their cultural/geographical differences, all have a very common way of thinking and viewing things around them, as if they were all growing within one single family or using the same brain. They can laugh at jokes that only they can understand and they can even understand

each other with simple words as if using telepathy. I understood what my advisor meant by that sentence and it irritated me because it was not a true sentence when viewed out of its context. In mathematics, we don't state a fact if there is an exception to that fact (even if one exception exists for billions of billions of billions of . of true cases, the fact still remains false and therefore cannot be stated as true). The fact that was true is that I was willing to give a free lunch in the absolute sense, and therefore the sentence stated by my advisor was false in the mathematical sense and should not be stated.

However my advisor was not using that sentence in the "street-language" sense. What he meant by it is the fact that complexity in mathematics never vanishes. There is an equivalent principle in physics that says "No creation ex nihilo". This means everything that exists comes from something that existed before, that has grown, or fragmented, or changed form. It is like the conservation of energy; energy can be transformed to various forms, but it can never be created nor destroyed. The "No free lunch" simply meant that complexity in mathematics cannot be destroyed. It can only be hidden (encapsulated) inside other mathematical objects. In other terms, when you solve a complex problem in which there is enough quantity of complexity and that the solution looks simple and elegant, this can only happen if one of the mathematical objects used in the solution is actually hiding the complexity, because there is no way you can make complexity vanish and disappear into non-existence. Therefore, whenever a mathematician (like my advisor) sees a simple solution, the first reaction would be to apply the "No free lunch" principle and try to find out the object (or objects) that are encapsulating the complexity that seems to have vanished by the elegant and simple solution.

The "No free lunch" principle applies not only in mathematics, but it is a valid principle in computer science too. We will use this principle in the design of a service-oriented-architecture to explain why the SOA Fabric (the messaging environment used in SOA) contains enough functionality so that the services and applications connected to it have less work to do.

3 "Asynchronous" is the force, and may the Force be with you

In this section we will try to explain the power of using asynchronous computing instead of synchronous. At first glance, it seems very restrictive to only use asynchronous. Someone would say "Why on earth would I restrict my architecture to use only asynchronous exchanges?" Before we answer this question, let us first understand the concept of "generic" versus "individual case":

Anyone who studied theoretical computer science at the undergraduate level has come across abstract data structures, one of which is a binary tree. If someone were to say that he will only use binary trees instead of using arbitrary tree structures, it would seem at first a very restrictive approach. As it turns out, there is a theorem in mathematics that proves that binary trees are generic among the collection of all trees. This means that any data structure represented by an arbitrary tree could also be represented by a binary tree. In

other words, binary trees by themselves could represent any data structure that arbitrary trees represent. Hence, a binary tree, even though it is just an individual case among all possible tree structures, a binary tree is actually generic enough to cover all the power of all the other trees. Therefore by working with only binary trees, not only the approach becomes simpler, but also all the cases that are covered by arbitrary trees are also covered the binary tree approach. This answers the question above “Why on earth should I restrict my architecture to asynchronous exchange only?” Similar to binary trees, it happens that asynchronous exchanges can solve any problem and scenarios covered by the synchronous exchanges. Therefore, using asynchronous exchange only is not in fact a restriction at all. On the contrary, it turns out that asynchronous exchanges simplify many things and the solution to complex problems becomes simpler when only asynchronous exchange is used. Asynchronous is really “The Power” behind SOA solutions.

Awareness of the power of playing asynchronous is growing more and more in the IT. For example, the groups defining web services standards have recognized that loosely coupled asynchronous processing is a more appropriate model for describing interfaces and interactions between applications. SOAP 1.2 and WSDL 2.0 have removed all dependencies on RPC-style invocations that were once heavily ingrained in their invocation models (they still support it but don’t promote it as the recommendation means of interaction). WS-BPEL and WS-Choreography are also based on a loosely coupled asynchronous exchange of information. Even JAX-RPC (the java specification for performing RPC-style invocations) decided to adopt asynchronous processing for Version 2.0.

Also, it turns out that loose coupling is much easily achieved when only asynchronous exchanges are used.

4 Loose Coupling achieved with asynchronous exchanges

Remote Procedure Call (RPC)-style programming has had reasonable adoption in the industry for a number of years (CORBA, RMI, DCOM, ActiveX, Sun-RPC, JAX-RPC, and SOAP). RPC-style communications tend to be synchronous in nature. By design, RPC-style programming mimics the serial thread of execution that a “normal” non-distributed application would use, where each statement is executed in sequence.

In tightly coupled RPC environment, each application needs to know the intimate details of how every other application wants to be communicated with—the number of methods it exposes and the details of the parameters each method accepts. As the number of applications and services increases, the number of interfaces that need to be created and maintained quickly unwieldy.

A simple formula often used for estimating the number of interfaces between applications is $n(n - 1)/2$. If the number of applications is 5, the number of interfaces is 10 (not too bad). But if the number of applications is 100, the number of interfaces is 4,950. This formula makes two assumptions: (a) it assumes that each application endpoint has only one interface, and (b) it assumes that every application needs to interact directly with every other application. In practice, each application will have more than one interface, and not

every application will need to interface with every other application. For example, if each application has an average of four RPC-style interfaces, and each of your 100 applications need to communicate with only 50% of them, your total number of interfaces will be 19,900. That's a lot of interfaces to create and maintain over time.

Loosely coupled interactions are made possible by using asynchronous communications. Asynchronous messaging allows each communication operation between two processes to be a self-contained, standalone unit of work. Each participant in a multi-step business process flow need only be concerned with ensuring that it can send a message to the messaging system.

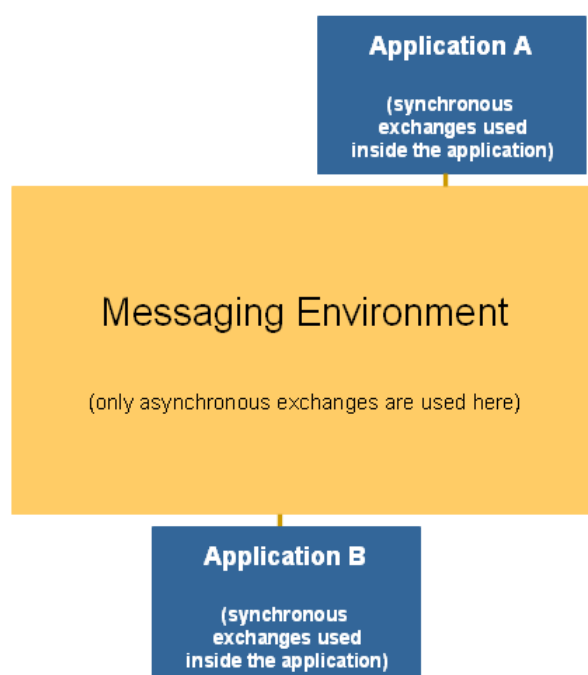


Figure 1: Asynchronous processing in a messaging environment

As the picture above depicts, applications are still allowed and encouraged to use synchronous exchanges, but only within the boundaries of the application itself and not when interacting with other remote applications. Asynchronous interactions are a key design pattern in loosely coupled interfaces. In an environment where multiple applications and services need to interact with each other, it is not practical that every application be required to know the details of every other application's myriad method calls and parameters.

An asynchronous message should be an autonomous unit of work that carries data and context around with it from place to place. In contrast to the $n(n - 1)/2$ problem of tightly coupled RPC, using loosely coupled messaging reduces the number of interfaces to something more linear, where the number of interfaces is exactly the number of connection points.

The service-oriented architecture builds on this brilliant idea of “loosely coupled asynchronous processing” by adding more functionality to the messaging system being used. The idea of adding functionality to the messaging system is a way of taking the work load off the applications that are to be connected to the messaging system. This is the second brilliant idea after the fact of playing asynchronous. This brilliant idea is just a simple consequence of the “No free lunch” principle: Let us assume this scenario. We have a hundred or more of applications (geographically very distant from each other) that need to be integrated in one big and complex system that would accomplish complex tasks. By definition, there is a big amount of complexity in this system that we want to build, and by the “No free lunch” principle, we can never destroy that complexity. We can only hide it. The fact of the matter is that the applications we want to integrate do not want to bear any amount of complexity because they already have their own complexity. Since we have to hide the complexity somewhere, and since the applications refused to have their share of this complexity, only one thing remains logically: the messaging environment. There is always a messaging environment when distant applications need to communicate with each others. The messaging environment is none but the system made of the components that help with the communication (with the exception of the parties themselves which are the applications we want to integrate). This is what service-orientation architecture is doing: applying the “No free lunch” principle, yield the conclusion that the messaging environment should have almost all the functionality. In other words, a service-oriented architecture pushes functionality (and complexity) to be handled by the messaging environment instead of being handled by the applications that are to be connected. This is similar to the plumbing in a house. Instead of having the plumbing everywhere in the house (in every room and every place), we tend to hide the plumbing either in the ceiling, basement or underground. The messaging environment of SOA is the place where all the plumbing is put together with complexity. This makes developing services easy as well as consuming them and integrating applications even easier.

5 Service-Oriented Architecture

The messaging system used by a service-oriented architecture is termed “SOA Fabric”. Other people sometimes refer to it by the name ESB (Enterprise Service Bus). What is the origin of this name? The word bus is actually coming from a hardware parlance. Inside every computer hardware there is an electronic piece called “Mother-board”. A PC mother-board has multiple slots that any type of compatible card can be plugged into. The use of the term “bus” in ESB is analogous to that. This is just to say that the messaging environment of the SOA architecture has a standard way that you plug into. This is the meaning of ESB (just to underline the standard way of plugging into the messaging environment of a service-oriented architecture). Since, ESB may relate (and have a connotation) to the way it is currently implemented (using combinations of MOM products and web services), in theory an ESB does not have to be implemented the way current implementations are doing. For this reason, to stay abstract and independent of how the messaging environment

is actually implemented (which technologies are used), we prefer to use the term “SOA Fabric” or “Fabric” for short as this term does not bear any implementation details.

The basic properties of a service-oriented architecture are the following:

- Plugging to the SOA Fabric is done in a standard way, through adapters called “SOA Service Containers“, or “Service Containers“ for short (SC).
- Whatever is being plugged to the SOA Fabric is not concerned with how it communicates with anything else that is plugged to the SOA Fabric.
- Anything that wants to plug into the SOA Fabric is concerned with only three operations: plugging into the SOA Fabric, posting data to the Fabric, and receiving data from the Fabric.
- The SOA Fabric gets the data to the applications (plugged into the Fabric) in the target data formats.
- What circulate inside the SOA Fabric are XML messages having a canonical standard form. An SC (Service Container) knows how to transform the canonical standard form of the Fabric to/from the target form understood by the application hooked to the SC.
- The SOA Fabric gets the data to the places that it needs to get to, in the target data format that it needs to be in. In a complex interaction across multiple processes and services, the response may not even go to the original requestor. Each response may actually be a new message being sent to a forwarding address.

An “SOA Service Container”, the adapter box through which applications can plug into the SOA Fabric, usually support many transport protocols (TCP/IP, HTTP, SMTP, etc). These transport protocols are like channel pipelines connecting the SOA Fabric with a Service Container box (SC).

The immediate observable benefits of the above architecture are the following:

- Loose Coupling is realized. The applications and services plugged into the SOA Fabric are loosely coupled in the sense that they all communicate using asynchronous messages that are sent to the SOA Fabric.
- Data Transformations are reduced from $2(n^2 - n)$ to $2n$. For example, suppose we have n parties and that each one of these n parties wants to communicate with every other party (every pair combination). For party A to communicate with party B, party A must know how to format messages in the form that is understood by party B, as well as be able to transform responses from party B into a format that is used internally by party A. This makes $2(n^2 - n)$ data transformations. If on the other hand, we let all the n parties plug into the SOA Fabric, then the number of data transformations would be reduced to only $2n$ in the worse case scenario. This is because each party would only care about being able to transform his own message

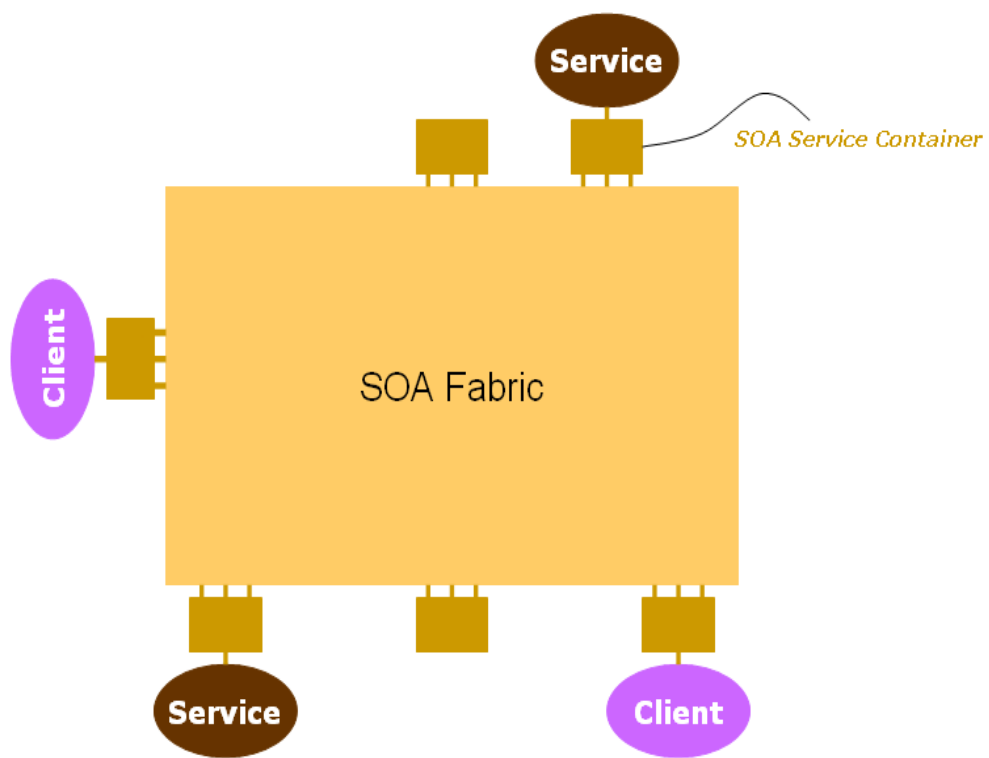


Figure 2: Service-Oriented Architecture

to the canonical standard form that circulates inside the SOA Fabric, and be able to transform from the canonical standard form of the SOA Fabric to the internal format used by the party.

- The SOA architecture would also most of the time reduce the $2n$ data transformations to zero. This is because the SC (Service Container) would be provided by many vendors, and along with the SC comes the ability to transform between the canonical standard message format of the SOA Fabric to various other types of format (since vendors compete with each other, every vendor would like to include support for many types of formats, that is be able to transform between various formats to the standard format used by the SOA Fabric). In other words, when an application or a service is deployed inside an SC (a service container), the SC would usually provide the data transformation between the SOA Fabric format and the message format as used by the application and/or service. For example, for 100 parties to communicate with each other without the SOA Fabric, they would have to perform 19,800 (Nineteen thousands and eight hundred transformations), but if they plug into the SOA Fabric, these parties will perform zero data transformations as each SC (Service Container) would be responsible for doing the two data transformations needed by a party (namely transformation between the standard format of the Fabric and internal format used by the party). Applications and services that will plug into the SOA Fabric will all be able to communicate with each other without maintaining the interfaces of each others and without doing data transformations. This is the best loose coupling that could be achieved.
- If services change their implementations (for example by upgrading to higher version), this would not break the interoperability between other parties. This is because the SC (Service Container) that lies on the side of the upgraded service will be responsible for transforming from the standard format of the SOA Fabric to the format used by the service, and therefore, such a transformation would take into consideration the upgrade of the service. Therefore, other parties would still continue to be able to talk to the upgraded service without even realizing that the service has actually changed or have been upgraded.
- Decoupling Transactions: "Transactions" is a very complex topic in software and it takes hard work and design to achieve. If we consider a distributed set of parties and that we want all of the parties to participate globally in one transaction, this becomes a nightmare in software because in this example, the complexities of transactions are augmented by the distributed nature of the parties and also by the fact that all of the parties are part of the same transaction (as opposed to a simple scenario where a transaction would involve two parties only). Without the SOA Fabric, solving such a scenario would be very hard and even harder to maintain as a solution since it would involve many proprietary techniques. If however, all the parties are plugged into the SOA Fabric, then global transactions become local transactions and this by itself is a huge reduction of the complexity. Having all message consumers and

producers participate in one global transaction defeats the purpose of a loosely coupled asynchronous messaging environment as provided by the SOA Fabric. In a loosely coupled environment, applications need to hand off their messages to the messaging environment using local transactions, and go about their business. For example party A would send a couple of messages to the SOA Fabric, the first of which would be "begin transaction" type of message and the last of which would be a "Commit" type of message. Party A continues his business without worrying about the transactional nature of the set of messages it has sent. The SOA Fabric, knowing that the set of messages it has received from party A are part of a transaction (all of them should succeed or none of them), it would make sure that all the messages (or none of them) arrive to the final target.

6 Service Orientation Principles

The services that participate in a Service-Oriented Architecture must satisfy at least four principles, which are the following:

6.1 First Principle: Explicit Service Boundaries

An SOA service must have an explicit boundary. The boundary surface represents what is visible outside of the service, and therefore could be shared by other services. The existence of explicit boundaries for services is to ensure the following:

- Services may be developed by teams spread over vast geographical/cultural distances. Service boundaries are meant to eliminate the cost of communication between developers. This means that each service could be developed independently of the other services with which it might interact in the future.
- Service boundaries also ensure that the implementation details of services do not matter. Services could be implemented in various languages/technologies and targeted to various platforms. Because of Service boundaries, services do not need to worry about how other services are implemented in order to interoperate with them.
- Since the boundary surface is meant to be small, cross-boundary communications are meant to be reduced as much as possible. If for example, two components depend too much on each other because they call each other all the time, the two components should be encapsulated within one single service instead of each component being a service by itself.

6.2 Second Principle: Service Autonomy

There are three aspects to "Service Autonomy":

1. The first aspect concerns the relationship between a client (that is a service consumer) and a given autonomous service. Prior to Service-Oriented paradigm, a service consumer (usually referred to by the term “Presentation Layer” in Software architectures) was usually distributed with the application that contains the service itself. In other terms, the consumer was part of the whole package which contains a given service and its backend resources. In Service-Oriented paradigm, the consumer is completely disconnected from a service. Consumers could be of various types, implemented in various technologies, and don’t care about the implementation details of the services they are meant to consume.
2. The second aspect of “Service Autonomy” concerns the relationship between two autonomous services. This basically describes the ability of an autonomous service to locate and negotiate communication between it and any other autonomous service. For example, service *A* calls service *B*. If Service *B* isn’t where it is expected, then Service *A* goes through a number of steps to discover Service *B*’s whereabouts. At no point does Service *A* require any information from Service *B* or any one else to try and complete the invoked method.
3. The third aspect of service autonomy is about service deployment. Contrary to the way applications are deployed with object-oriented architectures, service-oriented systems are deployed differently. In object-oriented world, an application is deployed as a unit (because of the dependencies between the objects that make the application). Service-Oriented Systems on the other hand, are usually deployed incrementally. New services are added to a given system without breaking the functionality, and services could even be deployed long time before deploying the service consumers. What makes this deployment scenario possible is the important fact that an “autonomous service is deployed in its own execution and security environment”

6.3 Third Principle: Shared Contracts and Schemas

In Object-oriented programming classes are convenient abstractions as they share both structure and behavior in a single named unit. Service-oriented development has no such construct. Rather, services interact based solely on schemas (for structures) and contracts (for behaviors). Every service advertises a contract that describes the structure of messages it can send and/or receive as well as some degree of ordering constraints over those messages. This strict separation between structure and behavior vastly simplifies deployment, as distributed object concepts such as marshal-by-value require a common execution and security environment which is in direct conflict with the goals of autonomous computing.

Services do not deal classes; rather, only with machine readable and verifiable descriptions of the legal “ins and outs” the service supports. The emphasis on machine verifiability and validation is important given the inherently distributed nature of how a service-oriented application is developed and deployed.

6.4 Fourth Principle: Policy-based Compatibility

Object-oriented designs often confuse structural compatibility with semantic compatibility. Service-orientation deals with these two axes separately. Structural compatibility is based on contract and schema and can be validated (if not enforced) by machine-based techniques (such as packet-sniffing, validating firewalls). Semantic compatibility is based on explicit statements of capabilities and requirements in the form of policy. Every service advertises its capabilities and requirements in the form of a machine-readable policy expression. Policy expressions indicate which conditions and guarantees (called assertions) must hold true to enable the normal operation of the service.