Joseph Fourier University

Master 2nd Year in Computer Science

(Software Architecture and Databases)

Title:

Protocol Oriented Service Deployment Platform

Author:
Anis Benyelloul (anis.benyelloul [at] gmail [dot] com)
Joseph Fourier University (Grenoble1), Grenoble, France

Supervisor:
Ishikawa Fuyuki (f-ishikawa [at] nii.ac.jp)
National Institute of Informatics(NII), Tokyo, Japan

SCHOOL YEAR 2007/2008

Protocol Oriented Service Deployment Platform

Author: Anis Benyelloul
anis.benyelloul [at] gmail [dot] com
Joseph Fourier University, Grenoble, France

Supervisor:
Ishikawa Fuyuki
f-ishikawa [at] nii.ac.jp
National Institute of Informatics (NII), Tokyo, Japan

School Year 2007/2008

Contents

List of Figures

Abstract

Service Oriented Computing is gaining momentum and interest both in the industry and in the research domain as the paradigm for the next generation of business software architecture. In this context, one particular aspect has been receiving much attention lately: *service protocols*. Service protocols are *enhanced interface descriptions* that contain not only the enumeration of the possible messages that a service can send and receive but also the *sequencing constrains* that must apply for a conversation to be valid. The purpose of this work is to show how the idea of service protocols can be leveraged at *runtime*, and how to design a "protocol oriented service deployment platform" that implements those concepts.

Résumé

L'architecture orienté services (SOA) intéresse de plus en plus l'industrie et le domaine de la recherche comme paradigme pour la prochaine génération d'architecture logiciel de gestion d'entreprise. Dans ce contexte, un aspect particulier reoit une attention particuliere : *les protocoles de services*. Les protocoles de service sont des descriptions augmentées d'interface qui contiennent non seulement l'énumération des messages possibles qu'un service peut envoyer et recevoir mais galement les *contraintes d'ordonnancement* qui doivent s'appliquer pour qu'une conversation avec le service soit valide. L'objectif de ce le travail est de montrer comment l'idée des protocoles de service peut être appliquée au moment de d'exécution, et comment concevoir une plate-forme de déploiement de service orientée protocoles mettant en application ces concepts.

Chapter 1

Introduction to Service Oriented Computing

Contents

1.1 Introduction

Nowadays, and more then ever before, business IT is put under high competitive pressure, and software systems are now required to adapt quickly and meet the requirements of an ever growing and fast changing market climate. Most of today existing infrastructures were not meant to cope with rapid adaptation and restructuring requirements, which adds another level of intricacy to an already complex IT landscape[10].

Specifically, the two most important requirements that modern IT are required to meet are: *integration* and *infrastructure management*. Integration means being able to efficiently combine and reuse existing assets both inside and across enterprise boundaries to create business value, despite the fact that all the individual parts have never been meant to be used together, use different conventions, are based on different platforms, etc. Infrastructure

management addresses two objectives: *Automation* and *virtualization* of the environment. *Automation* aims at keeping the necessary human intervention to a minimum, and reduce management complexity to enable better use of assets. Environment virtualization aims at making all enterprise assets easily accessible from anywhere, just like if there were local[10].

In this context, Service Oriented Architecture (SOA) is perceived by many as the software engineering paradigm for the next generation of business software. One that will permit rapid development, as well as fast re-configuration and re-composition of existing software assets.

On the other hand, it seems that the most important hurdle that could prevent wide adoption of SOA is actually a lack of understanding about the real concept behind it. The next sections will try to show why, as well as give some insight to help establishing a better understanding of the most fundamental ideas.

We will first try to give a general overview by focusing on what's really new about the concept of SOA. After that, we'll take a step back and consider each concept and characteristic in more detail.

1.2 SOA, Beyond the hype

1.2.1 Lack of focus

It seems that the main source of misunderstandings about SOA comes from the fact that most introductory resources on the subject put the focus on the wrong concepts or properties, or do not insist enough on the actually relevant and distinguishing concepts that underlie SOA.

Many resources try to define SOA by defining what a service is, and they often end up with statements such as:

> A service is the fundamental, and primitive component of SOA.
> A service fulfills a function that is well-defined, self-contained, and does not depend on the context or state of other services.

This is correct, but the same definition could apply to the classical "procedure" of the procedural programming paradigms of the 70s, and thus it does not tell much about what SOA is.

Other possible definitions of SOA include:

> SOA is about decomposing applications into loosely-coupled parts (called services) that have a clearly defined interface which promotes evolution, reusability and composability.

Again, this is true but not different from what Object Oriented Programming has been trying to achieve since its introduction in the 80s.

Other papers yet introduce *Web services* along with the definition of SOA. But as many authors have already noted [9]:

- There is nothing new about the concept of Web services, it is just a new form of distributed computing based on open standard (like CORBA).
- When it comes to SOA, Web Services are just one possible implementation platform, and certainly not part of the concept of SOA itself.

Some resources also try to include business process management concepts into SOA:

> SOA purpose is the control business processes, establishing corporate-wide security, privacy, and implementation policies, and providing auditable information trails, are all examples of ways that SOA can reduce several of the risks facing companies today.

The companies that actually didn't have any form of privacy, security and implementation policies, even long before the advent of SOA, were very seldom. Besides, business process modeling is just one possible application domain of SOA, and again not part of the concept of SOA itself.

All in all, among the most common mistake we make when trying to introduce SOA is to put the focus on:

Properties of Service Oriented Software such as loose-coupling, reusability, clear interface/implementation separation, abstraction... these are actually the well known properties of "good" software design that are carried over to the SOA world, but they should not be used as a foundation for understanding SOA.

Web Services Web services is just one possible *implementation platform* for SOA, and does not bring any new ground breaking ideas compared to existing distributed application middlewares[9].

Business Process Management Which is one possible application domain for SOA, and thus should not serve as a base for explaining what SOA is.

As a result of this confusion some authors arrived at the conclusion that there is nothing really new about the concept of SOA, that it is just a "hype" or a yet another technology buzzword[11].

1.2.2 Service Oriented Architecture

In order to precisely yet fully understand the concept, it useful to split the problem into two parts: the objective of SOA and the approach of SOA.

The Objective

The "objective" is what SOA strive to achieve. The visionary promise of SOA is a world were one could develop new applications by leveraging existing ones. This concept comes as the logical evolution of previous re-use efforts that were first trying to reuse procedures, then procedure libraries (e.g numerical analysis libraries), then classes and class libraries (e.g GUI toolkits), then came the component based software development (CBSD) paradigm which promotes the idea of being able to build software by *easily* assembling existing components without having to write much code yourself. The next step in this evolution is to be able to reuse *entire applications*.

Re-using applications does no mean that you "download" them (or part of them) and link them into your own (this would be traditional class library or CBSD reuse). It means that the applications are already deployed and used, that their maintenance and evolution is under the responsibility of other people. One just has to leverage that application (where is it) and use its functionalities for his own purposes.

The fact that you can also reuse application across organization boundary, means that your own software would be composed of parts that are actively run and under control of third parties. This means that application development time can be dramatically reduced (you now have a much larger scope of existing software to choose from). But this obviously comes at the cost of a lot of issues (both technical and non-technical) that are beyond the scope of this work.

SOA is also different from traditional distributed computing. In the sens that, in the latter, you create an application that you "*split*" across a network

of hosts. SOA adopts almost the opposite approach where you create new applications by "*assembling*" remotely available functionality.

The Approach

The "approach" defines *how* does SOA try to achieve its objective. This is where the notion of "service" come in. In order to be able to re-use existing applications they must *expose* their functionality as externally visible "services", which are simply a set of "*operations*" or "*functions*" (function names and arguments). This means that software developers must now integrate the idea that their applications could be remotely re-used by other ones, and thus make their design in a such a way that they can expose key functionalities as services.

1.2.3 Wrap up

In order to make the definition more comprehensive we can add the following points:

- Web services are currently the most common and widely adopted implementation platform for SOA software systems. They define a common set of file and message format, that enables the inter-operability of applications running on different environments/platforms (e.g integrating a Java EE application with a .Net one)
- The most promising application domain for SOA seems to be business process modeling. Where each service becomes the entry point for the implementation of a business process.
- The fact of having to think of applications as composed and from different other applications that are run and maintained by third parties (possibly in an other organization) actually implies that the interfaces between the applications need to be designed in a way that permits the application evolution without breaking the clients. Which in turn means that it will most likely lead to clean and well designed architectures: these are the good "side effects" of SOA.

From what has been seen, it is necessary to distinguish between the different ideas revolving around the notion of SOA if we want to be able to pin

point the essential concepts behind it, the ones that set it aside from the previous paradigms. This is essential to form a sound base of understanding that will enable both industry wide adoption and research development.

1.3 Fundamental SOA

Due to its recent introduction in the field, service oriented computing has been getting a lot of attention both from the research and from the business IT industry. As a result, a whole cloud of additional concept and properties often gets associated with the basic principle behind SOA (which is application re-use, within and across organization boundaries, as seen in the previous section).

This section will try to give more comprehensive overview of service orientation, by detailing the most common "SOA good practices" and "design guide lines" as well as giving the most common design properties that every proper service oriented architecture has.

1.3.1 An analogy

In order to understand the idea behind service orientation, it is useful to take the analogy of nowadays business companies. Business companies rarely work in isolation, most of them will have to interact with many other companies in order to produce value. For example, a "farm" company that grow apples need to form relationships with e.g a fertilizer company, from which it can buy fertilizer, in addition it will need a relationship with e.g a jam factory company, to which it may sell its products. The jam company will then need an agreement with a distribution company to bring its content to the market, and maybe a separate advertisement company to make its products known to the public.

In order to create healthy and durable relationships that are profitable for both parties, companies need to setup a set of agreed upon "*conventions*" and standards (for example a common currency for the exchange of goods and services) for their interaction, most of the time they also have to both agree on a "*contract*" that defines the commitments that both parties make as well as the expectation each party hold upon the other as far as their interaction is concerned.

Each company tries to strike a balance between the following two conflicting drives:

- The need to retain its independence, to be able to evolve freely and easily choose a different partner company whenever the opportunity arises.
- The need to make each interaction and partnership as effective as possible. This may lead the company to try to adapt some of its infrastructures to accommodates its current particular partner company.

The challenge is to create links that are tight enough to make the partnership efficient, but still loose enough so that, at a global level, the business company market retains its dynamic nature and its ability to adapt changes.

In essence, today's business companies form a network of interdependence. Each element in the network is completely self-managed and independent from the others, yet, it needs to interact with its neighbour elements in order to produce value. This is essentially the idea behind "service orientation".

Similarly to this analogy, SOA is about making each individual component of a big software system independent, self-managed and able to evolve independently from the others, but at the same time, in conformance with a certain set of standards and convention that will ensure the required interoperability of each component with its environment. In the context of service oriented architecture components are referred to as "*services*".

1.3.2 Services Are Units of Encapsulation

Each service encapsulates a certain "*processing logic*" that basically represents the task it is supposed to fulfill. The concern addressed by a service can be small (small granularity services, usually taking care of low level and/or purely technical aspects of a software system), or large (coarse granularity services, usually representing high level concepts such as entire business processes).

Further more, service logic can encompass the logic provided by other services, in which case it is referred to as a "*compound service*".

As shown in Figure 1.1, when building an automation solution consisting of services, each service can encapsulate a task performed by an individual

step or a sub-process comprised of a set of steps. A service can even encapsulate the entire process logic. In the latter two cases, the larger scope represented by the services may encompass the logic encapsulated by other services.

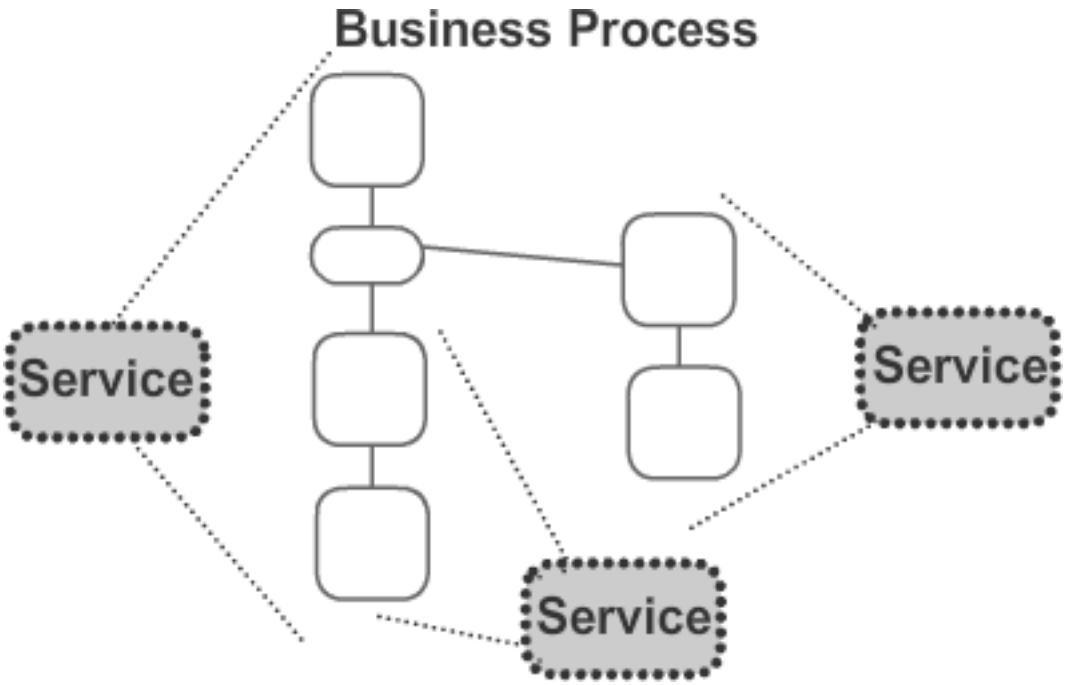

Figure 1.1: Services can encapsulate various amounts of logic.

1.3.3 How services interact

Individual services can be used by "client programs" or by other services. In order for an entity (be it a service itself or not) to interact correctly with a service, it needs to be aware of it. The required information to use a service is contained in its "*service description*".

The purpose of a *service description* is to provide information about the interface of a service to potential service users. At the very least, this should include the name of all the supported operations, the parameters and data types expected and returned by a service.

The fact that services are able to "describe themselves" by providing their interface description to potential clients results in a loose coupled relation

between the service and its users. For example, Figure 1.2 illustrates that service A is aware of service B because service A is in possession of service B's service description.

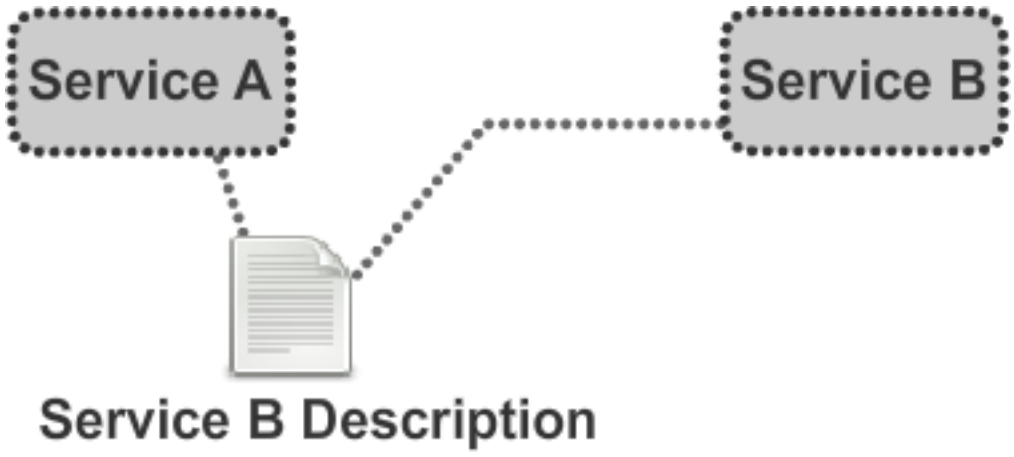

Figure 1.2: Because it has access to service B's service description, service A has all of the information it needs to communicate with service B

1.3.4 How services communicate

Service interactions imply data exchange. Thus, it is necessary to define a communication framework allowing easy interoperability between services while preserving the loosely coupled quality of service's interactions. One such framework is "*messaging*".

Messages are independent units of communication that can carry all sort of "meta-information" (in addition to the message content itself) that will permit to implement many communication-related features such as quality of service properties (see Figure 1.3).

1.3.5 How services are designed

So far, we've described SOA as a set of services exposing their service description to others and using a common messaging framework to communicate. This forms the basic architecture. The distinguishing aspects of SOA that makes this different from traditional distributed architecture that support messaging and a separation of interface from processing logic, is how the

Figure 1.3: The message is the basic unit of communication in SOA

three core components (services, descriptions, and messages) are designed. This is where "*service orientation*" comes into play.

Much like object-orientation, service-orientation has become a distinct design approach which introduces commonly accepted principles that govern the positioning and design of our architectural components (Figure 1.4).

The application of service-orientation principles to processing logic results in standardized service-oriented processing logic. When a solution is comprised of units of service-oriented processing logic, it becomes what we refer to as a service-oriented solution.

The key principles of service orientation are:

Loose coupling Services maintain a relationship that minimizes dependencies and only requires that they retain an awareness of each other.

Service contract Services adhere to a communications agreement, as defined collectively by one or more service descriptions and related documents.

Autonomy Services have control over the logic they encapsulate.

Abstraction Beyond what is described in the service contract, services hide logic from the outside world.

Reusability Logic is divided into services with the intention of promoting reuse.

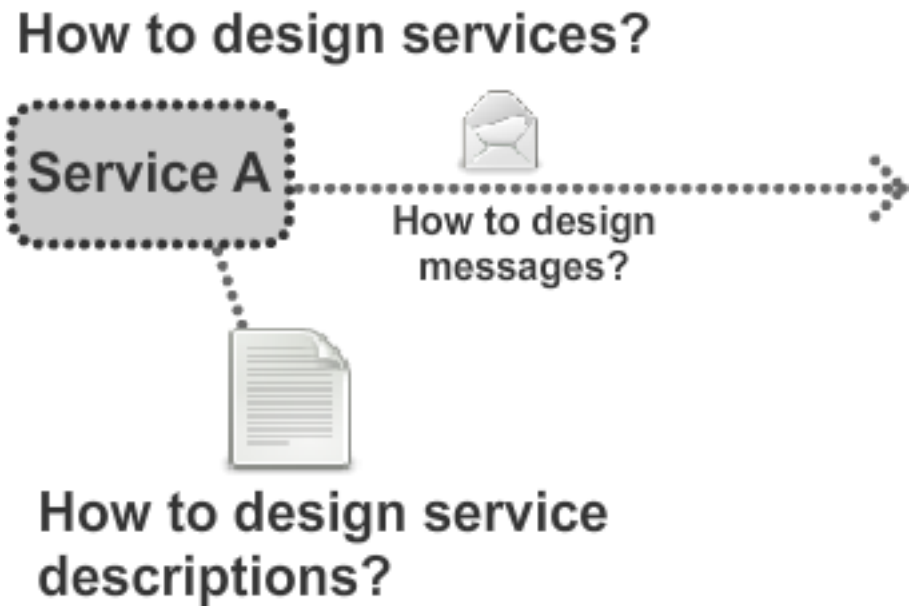

Figure 1.4: Service orientation is about how to design the services, the descriptions and the messages

Composability Collections of services can be coordinated and assembled to form composite services.

Statelessness Services minimize retaining information specific to an activity.

Discoverability Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.

Now that we have defined the three core components of SOA (services, descriptions and messages), and a set of principles that should guide the design of those components, all that is missing an implementation platform that will allow us to pull these pieces together to build service-oriented automation solutions. The "*Web service*" technology set and standards offers such a platform.

1.4 SOA in practice

The industry world has had a great influence on SOA. As new practical problems arouse, new solutions had to be invented and put to practice. The fundamental SOA principles remain, but those requirement-driven evolutions have shaped the real world look of SOA.

Major software vendors play an important role in this context. As they are constantly addressing real world problems, and creating increasingly powerful XML and Web service support into their product platforms. The result is an extended variation of service oriented architecture we refer to as "*contemporary SOA*".

Although the actual implementations and feature set vary from vendor to vendor (and from vendor product to the other), contemporary SOA has come to a point where a set of common characteristics start to become the "*de facto*" standard. Exploring those characteristics will give a better idea of how nowadays IT businesses are actually leveraging the fundamental aspects of SOA to respond to their infrastructure and evolution requirements.

Specifically, we explore the following primary characteristics. Contemporary SOA:

- Addresses quality of service issues;
- Is fundamentally autonomous;
- Is based on open standards;
- Supports vendor diversity;
- Promotes service discovery;

1.4.1 Contemporary SOA addresses quality of service issues

Enterprises tend to have strong reliability requirements regarding their software infrastructures. Thus, there has been many efforts to bring quality of service issues (reliability, safety, speed) into SOA.

The most important quality of service issues are:

Security Includes the protection of messages (encryption) as well as setting up permission policies for the use of individual services (e.g through the use of "accounts" and "login" operations in services)

Quality of message delivery Includes features to maximise the probability that a message reaches its final destination, as well as exception mechanism to notify a service about the failure in a delivery.

Performance Includes features to make sure the overhead imposed by XML (and derived standards: SOAP, WSDL, etc) does not become a performance bottle-neck.

Transactions Are essential to any business activity to protect the integrity and coherence of data in case of exceptions.

Many of such requirements are addressed by individual web service extension standards and specifications. The ultimate objective being to enhance the primitive SOA model with "quality of service" capabilities.

1.4.2 Contemporary SOA is fundamentally autonomous

Autonomy is one of the fundamental concepts of contemporary SOA. And it is realized in the following ways:

- Individual services are as independent and as self-contained as possible. This means that they retain total control over the underlying application logic they represent.
- Messages are loaded with meta information (such as quality of service information) that dictates how they should be handled by the recipients. This makes messages "autonomous units of communication".
- Enterprise level applications composed of many independent services can themselves be perceived as autonomous services responsible of their own business logic within the enterprise boundary.

By creating service abstraction layers, entire domains of solution logic can achieve control over their respective areas of governance. This establishes a level of autonomy that can cross solution boundaries.

1.4.3 Contemporary SOA is based on open standards

As seen previously there is nothing fundamentally new behind the idea of web services ([9]). It is just a form of remote procedure call, like RMI,

or RPC (web services originally started as a variation of RPC using XML: XML-RPC).

The most important contribution behind web services, is not the fundamental idea behind it. It is the fact the they are a set of standards that all vendors seem to have adopted. The set of standards define a complete framework for the implementation of remote procedure calls. From the definition of the interfaces (WSDL) to the way data is transmitted and messages are formatted (SOAP).

Contemporary SOA takes this concept to the next level, and actively limits the role and the impact of any proprietary technology or format can have on the implementation of the service structure(Figure 1.5).

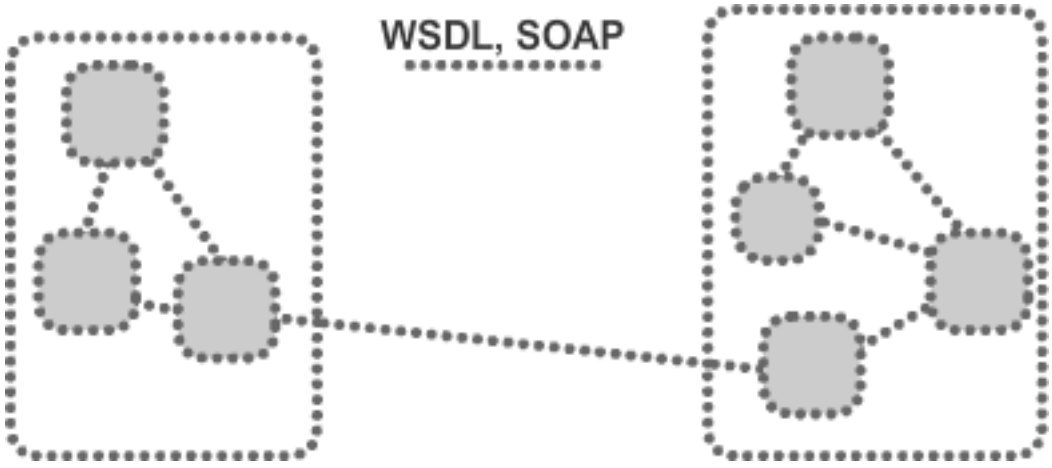

Figure 1.5: Open standards and technologies are used inside and outside solution boundaries

1.4.4 Contemporary SOA supports vendor diversity

Closely tied to the concept of open standards discussed in the previous section is vendor diversity. The fact that enterprises rely exclusively on open standards for their business ITs not only means that they can interoperate with other ITs using the same set of standards but also that they now have the freedom to choose any application vendor that will match the needs of their specific application.

Web services serve as a access point to applications running on other platforms. No matter how proprietary a development environment is, the fact

that it can present it's functionalities to the outside world as web services, means that it can interoperate with applications running on other development environments (Figure 1.6).

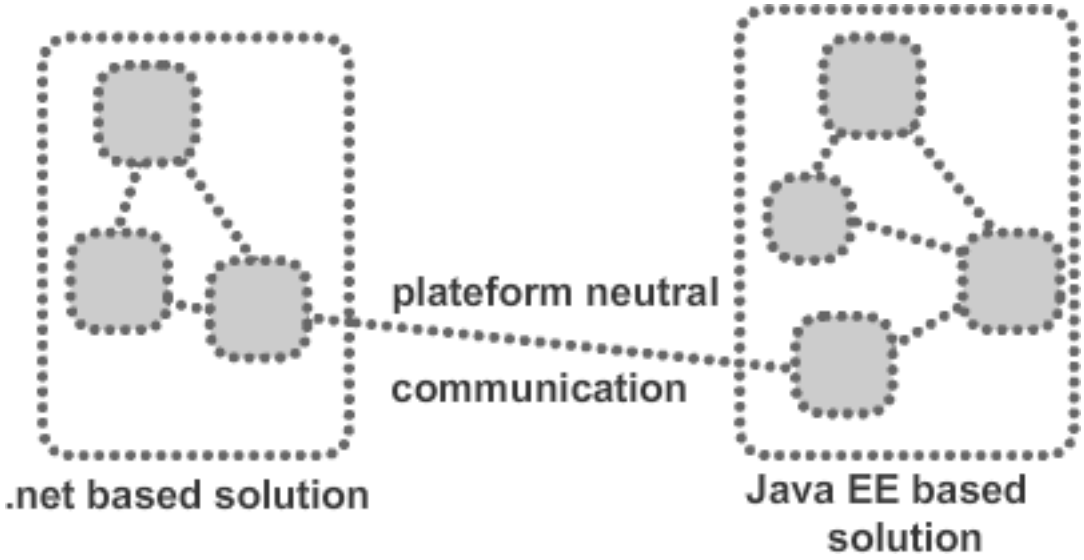

Figure 1.6: Thanks to open standards inter-operation across platform becomes possible.

This also means that organizations can choose to concentrate on their existing assets and tools, while still leveraging the advantages of SOA. On the other hand the option of switching to other vendors is still there. The web service framework provides a common base everyone agrees upon and relies on. It provides the common language that allows applications to inter operate despite a heterogeneous vendor environment.

1.4.5 Contemporary SOA promotes service discovery

Even though the concept of service discovery was given a lot of attention, it hasn't convinced many enterprises. Part of the reason might be that there are simply not enough alternatives to any particular service in a certain context (e.g if an application needs to contact a "resource management" service to monitor how much of a certain resource is available to the enterprise, there is often only on such service that is responsible for that information, and there is little sens in making the application "discover" or "look for it" because it

is well known in advance). Also, many initial uses of the web services platform were inspired by the more traditional distributed computing mind set and involved only simple point-to-point connections. These were essentially implementations of distributed computing environment (not SOA) using web services as the implementation platform.

SOA encourages the concepts of service registry and service discovery. Many cutting edge SOA will be likely to rely on some form of service discovery whenever appropriate.

Chapter 2

Service Protocols: Enhanced Interface Specifications

Contents

2.1 Introduction

One particular concept that has recently been gaining interest in the research domain is including, as part of the service description, not only it's interface but also it's *business protocol* or *service protocol*.

In the traditional model, service description only consists of "interface descriptions" (often in the form of WSDL files) that enumerates the set of operations supported by a service, and for each operation/method, it's name, number and type of arguments, as well as information about the return value (if any) and error reporting.

The traditional model is known to have some shortcoming, among which:

- There are often "implicit" rules that govern how the interface might be used, and when each of its methods can be invoked. For example, in the case of an e-shopping service, it makes little sens to call the *checkout()* method as a first operation, before having logged in or added anything to the cart. These rules are often documented in a separate "user

manual" and in an informal language. But they can't be represented using the current web service technologies (such as WSDL).

- The traditional model often implies a "*client-server*" vision, where one service (or more generally, one "end point") acts as the "user" of another. One service is always the one "*requesting*" and the other "*responding*" to the requests. For example, the traditional interface descriptions concern only messages that can be *sent* to the service, and not about messages (or request) that the service may send.

- The idea that both services can send requests to each other brings about the fact that we need to know how does the messages entwine to create meaningful conversations.

The idea behind service protocols is to adopt "*augmented interface descriptions*" containing:

- Message/method signatures both for messages sent and messages received, and

- Protocols defining the legal sequences of messages that can be exchanged between a service and its mate.

A service protocol can be thought of as a *formal* "user manual" for the service interface: it gives instructions on how to use it. Many times, the different methods a service exposes have a certain semantic that bind them together, such that it is not possible to invoke anyone of them anytime. More often than not, there will be strict rules and constraints that has to be followed by the service user in order to make correct use of the service interface. That information is the realm of service protocols.

We consider that service protocols are bidirectional, describing both messages that can be sent and messages that can be received by a service.

Closely related to the notion of service protocol is the notion of "*protocol compatibility*" such that protocol-compatible services can be determined to be free of certain important errors that cannot be caught by the type system alone. In the following we'll describe a simple method to check if two protocols are compatible and thus if it is possible to compose the two corresponding services.

Integrating the service protocols as part of the service oriented technologies brings many important applications that can considerably simplify and support web service development, debugging and maintenance.

For the service user :

Formal interface specification The service user now has more detailed information on how to use a service correctly and is less likely to develop erroneous clients. Runtime time tools can be used to check that clients' usage of a particular service is compliant with its protocol.

Extended search It is now possible to perform searches in service registries based on the supported protocols. Clients will thus receive only informations about service they can actually interact with.

For the service developer :

Automated code generation When the service developer wants to implement a new service based on an existing service protocol (e.g a new e-shopping service that supports a standard e-shopping protocol), it becomes possible to automatically generate a code skeleton for the service based on the protocol it is supposed to implement. Just like WSDL and IDL specifications can be used to obtain code templates.

Automated error handling Since the protocol tells when and under which circumstances each messages can be sent or received, the middleware can now automatically filter the incoming and outgoing messages and notify the service code of any "unexpected message".

Compliance test Services can now be automatically *tested* for compliance with standard protocol, to ensure that they will interact correctly with any client that supports that standard.

In order to put the idea of service protocols into practice, we need the following:

A Protocol Model : It is clear that the first step is to choose a formal"model" to represent protocols. Many possibilities exist and many have already been explored, among the most common we find "*Peri networks*" and "*Finite State Machines*" and their corresponding variations and extensions. In any case, the chosen model must be expressive

enough so that one can express the most common requirements that usually apply to service interfaces, but at the same time, simple enough so that we can process, check, and assemble protocols easily and without compromising the performances. In this work, the model we have chosen is finite state machines (as described in more details below), because they are at the same time very simple and easy to process but at the same time can represent the vast majority of requirements one might want to impose on service interfaces[5].

Protocol Operations : Given a certain protocol model, the next step is to define a set of basic operations to manipulate protocols. In the context of finite state machines, many protocol operators, and predicate exist and have been studied in previous works[1][12], among which:

Compatibility Analysis : This is the one that first comes to mind when thinking about protocols. "Compatibility analysis" aims at saying, given two protocols, if the corresponding services are valid "mates" of each other, that is, if they can actually engage in a conversation. Compatibility analysis sometimes also include the ability to pinpoint the exact "incompatibility spot(s)" between two protocols, which can help fix "almost compatible" protocols.

Repleceability Analysis : The objective of "replaceablility analysis" is to tell if it is "*safe*" to *replace* one protocol with another (or one version of a protocol with another) without breaking existing clients. In other words, given two protocols, say if one of them will accepts all the possible conversations that are accepted by the other. This kind of analysis is useful when updating existing services and their corresponding protocols.

Protocol compliance analysis : Here the objective is to compare an existing program (client or server) to see if it conforms to a given service protocol (i.e. if a client uses a service correctly, or if a server conforms an imposed protocol).

Tools : The "tools" are simply the implementation of the protocol management and analysis operations seen above, in the form of computer programs. Many of the currently existing tools are "*design time tools*", which means they aim at aiding the design and development of service clients and servers. The present work will concentrate on "*runtime*

tools" that will support the execution of services with service protocol features.

2.2 Service Protocols

We assume that services interact with each other via typed interfaces. An interface consists of two parts:

- The set of messages that can be send and received through it. This can be expressed with traditional interface description languages such as IDL or WSDL. It describes the set of messages that can be exchanged between the service and its mate. Besides indicating the type of its parameters, each message in a collaboration specification is labeled as a "*send*" messages (which means the service can send the message to other paty) or a "*receive*" message (which means the service is ready to receive the message from the other party).

- The corresponding protocol that specifies the allowed *sequences of message exchanges*. It describes a set of "*sequencing constraints*". Sequencing constraints define legal orderings of messages by means of a finite-state machine (see below).

A finite state machine consists of a set of states and transitions between those states. The states represent the different "phases" or "steps" in a given conversation the is service engaged in, while the transitions represent messages sent and received. Sending or receiving a message is what makes the conversation advances to the next step.

A transition will be represented as follows

$$< state >:< direction > (< message >) \rightarrow < state >$$

where:

- *state*: is the symbolic name of a state;

- *direction*: is either "*send*" or "*receive*"; and

- *message*: is the name of a message described in the interface signature.

Every protocol P has a unique state $init_P$ that is the initial state when the conversation begins. And might have any number (including zero) of end states from which it is not possible to advance to another state (there are no transitions emerging from end states).

We consider only "*deterministic*" protocols: we do not allow the same message to be able to trigger more than one possible transition. When two transition emanate from the same state, and are both *send* transitions or both *recieve* transitions, they must have different message labels, i.e if:

- $s_1 :< direction_1 > (M_1) \rightarrow s_2$
- $s_1 :< direction_2 > (M_2) \rightarrow s_3$

are both transitions and $< direction_1 >=< direction_2 >$ and $s_2 \neq s_3$, then $M_1 \neq M_2$. This is not much of a restriction however, because language theory has shown that it is possible to transform a *non-deterministic* finite state machine, that recognizes a certain language, into a *deterministic* finite state machine that recognizes the same language. That algorithm could apply in this case too, to transform a "*non-deterministic*" protocol that accepts certain sequences of messages into a corresponding "*deterministic*" protocol that accepts the same set of message sequences.

Initially all protocols will be in their respective initial states ($init_P$). At each step during the conversation, emanating transitions tell which messages might be sent, and which messages the service is ready to receive, i.e. Assuming the current state is s, a service is allowed to send a message M only if:

$$s : send(M) \rightarrow s'$$

is a transition in the protocol. Similarly, the service is ready to accept a message M only if:

$$s : receive(M) \rightarrow s'$$

is a transition. When a message is sent/received the protocol moves to the next state (s') and the interaction can continue in a similar fashion.

When a message is sent (or received) the associated parameters are transmitted as well. Note however, that the actual value of the parameter cannot influence the protocol advancement (in our model, it is not possible to describe a transition that depends on the value of a parameter). One the one hand, this restriction simplifies the protocol analysis, and on the other hand,

it is always possible to break a "parameter dependent transition" into multiple transitions with different message labels.

A protocol may have final states with no outgoing transitions, or it may be nonterminating. A state is local to an interface; each interface of a service can be in a different state.

A state in which messages can only be sent is called a *send state*. A state in which messages can only be received is called a *receive state*. A state in which the component can either receive or send a message is called a *mixed state*.

Note that each service may be simultaneously involved in several message exchanges with different clients, and therefore, will have to keep track of multiple concurrent instances of the protocol state machine.

We also assume that a service will not indefinitely remain in a state that contains a send transition. If it is a send state, or if it is a mixed state but does not receive a message, it will ultimately send a message from that state. Without this assumption, it might be that one party is in a state where it can send message m, and its mate is in a state where it can receive message m, but no progress is made, as the sender never sends the message.

2.2.1 Examples

This section introduces example of service protocols.

The first example is an RSS feed service. The purpose of the service is to provide its client with a feed of news about a certain topic:

```
RSS Feed Service {
    Receive Messages {
        Login(user, pass);
        Read(topic);
        Stop();
    }

    Send Messages {
        Denied();
        Ok();
        M();
        End();
    };
```

```
    Protocol {
        States {1,2,3,4};
        Transitions {
            1: receive(Login)-> 2;
            2: send(Denied)  -> 1;
            2: send(Ok)      -> 3;
            3: receive(read) -> 4;
            4: send(M) -> 4;
            4: send(End) -> 3;
            4: receive(Stop) -> 3;
        };
    };
};
```

Figure 2.1: RSS feed service protocol

See figure 2.1 for a graphical illustration of the finite state machine corresponding to the service protocol.

Initially the service is in state 1, in which the only acceptable operation is `Login`, which means the service client must first login before he can get access to the feed.

Once the `Login` message is sent, the service moves to state 2. In state 2 there are two possible transitions. Either going back to state 1, after having sent a `denied` message. Or onto state 3 after sending an `ok` confirmation message. This essentially means the service is checking the provided login information and will either reply positively and move on to state 3. Or will respond negatively and move back to state 1 to wait for another `Login` attempt.

Once in state 3 the service waits for a `read` message that will confirm that the client wants to start receiving the news. The service will then move to state 4.

In state 4, the service can either:

- Send a news message `M` and stay in state 4;
- Send a message `End` and go back to state 3. This will happen when there are no more news messages to send;
- Receive a `Stop` message from the client, which will cause the service to stop sending messages and go back to state 3.

The second example is an imaginary e-store. Clients of the service buy products by filling an electronic cart, choosing a payment method and finally placing their order:

```
E-Store Service {
    Receive Messages {
        AddToCart(product_id);
        Paym(payment_method);
        Check();
    }

    Send Messages {
        Denied();
        Ok();
    };

    Protocol {
        States {1,2,3,4,5};
        Transitions {
            1: receive(Paym)-> 2;
            1: receive(AddToCart)-> 1;
            2: send(Denied)  -> 1;
            2: send(Ok)      -> 3;
            3: receive(check) -> 4;
            4: send(Ok) -> 5;
            4: send(Denied) -> 3;
```

```
        };
    };
};
```

See figure 2.2 for a graphical representation of the protocol.

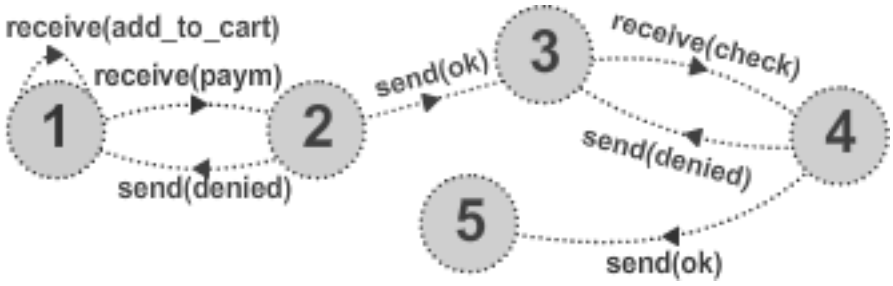

Figure 2.2: eStore service protocol

Initially the service is in state 1, in which the service can accept two operation: `AddToCart` to add a new product to the electronic cart, in which case the protocol goes back to state 1. Or it can accept `Paym` that signals that the user has finished adding products to the cart and wants to choose the payment method. In that case, the service goes to state 2.

State 2 contains only "send" transitions. From that point, the service can send its client either an `Ok`, in which case it means the payment method is supported, and the service goes to state 3. Or the service can send a `Denied` which means the payment method was rejected in which case the service goes back to state 2.

State 3 is where the user confirms his order by doing a "checkout". The service receives a `check` message and goes to state 4.

In state 4, the service will either send a `Denied` message, to notify that the payment did not went through correctly and go back to state 3, or it will send a `Ok` message to notify the user that his order has been confirmed.

Once the `Login` message is sent, the service moves to state 2. In state 2 there are two possible transitions. Either going back to state 1, after having sent a `denied` message. Or onto state 3 after sending an `ok` confirmation message. This essentially means the service is checking the provided login information and will either reply positively and move on to state 3. Or will respond negatively and move back to state 1 to wait for another `Login` attempt.

Once in state 3 the service waits for a `read` message that will confirm that the client wants to start receiving the news. The service will then move to state 4.

2.2.2 Service Protocol Semantics

This section present a formal definition of service protocols, protocol interactions, and shows how to implement protocol compatibility analysis.

There are two semantics one can assign to message exchange between collaborating services[12]:

Asynchronous semantics Under the asynchronous semantics each protocol advance independently from the other. A service may send a message m when it's protocol is in a state that permits to send a message m, even if the recipient of the message is not ready to receive the message (yet). Each service in this model will have to maintain a queue of messages that have arrived but haven't yet been "consumed". [3][6].

Synchronous semantics Under the synchronous semantics, a service S can only send a message m to its mate if S is in a state that enables it to send m and if its mate is in a state that enables it to receive m. The finite-state machines describing the protocols of the two services *advance synchronously*, so that the sending and receipt of a message are considered an atomic action under this abstraction. No queues are required. In the following we'll focus primarily on the synchronous semantics because it fits better with the notion of service protocols.

It is actually possible to implement the synchronous semantics without actually requiring the two services to send and receive messages atomically. It turns out that all we really require is that the two components always agree on the execution trace — the order of messages sent and received (c.f. 3.4).

We define a service protocol P as being a tuple

$$P = \langle States_P, Init_P, EndStates_P, Transitions_P, Messages_P \rangle$$

where:

- $States_P$ is a finite, non empty set of states;

- $Init_P \in States_P$ is the initial state of P;
- $EndStates_P \subset States_P$ is a set of end state (can be empty);
- $Messages_P$ is a finite state of messages. We define the function $Polarity(m, P)$ that associates a message and a protocol with a "direction", either $+$ for a receive message or $-$ for a send message.
- $Transitions_P \subset (States_P \times Messages_P \times States_P)$ is a set of transitions from one state to another in response to a certain message. As stated above we'll often represent transitions textually as:

  $$s : send/receive(m) \rightarrow s'$$

  where $m \in Messages_P$ and $s \in States_P$ and $s' \in States_P$.

In the rest of this description we make the assumption that given two protocols P_1 and P_2 the names of messages sent from P_1 to P_2 is disjoint from the names of messages sent from P_2 to P_1. This allows us to write a transition as $s : m \rightarrow s$, omitting the *send* and *receive* keywords from the message. This does not restrict the set of possible protocol collaboration that applies to this study, because it is always possible to *rename* the messages in such a way that the two message sets are disjoint.

A *collaboration state* for protocols P_1 and P_2 is a pair $\langle s, t \rangle$, where $s \in States_{P_1}$ and $t \in States_{P_2}$.

A *collaboration history* for protocols $P1$ and $P2$ is a possibly infinite sequence of the form $\alpha_1 \rightarrow_{m_1} \alpha_2 \rightarrow_{m_2} \ldots$ where

- each $\alpha_i = \langle s_i, t_i \rangle$ is a collaboration state of P_1 and P_2;
- $\alpha_1 = \langle init_{P_1}, init_{P_2} \rangle$;
- $\alpha_{i+1} = \langle s_{i+1}, t_{i+1} \rangle$ if and only if $\alpha_i = \langle s_i, t_i \rangle$ and $(s_i : m_i \rightarrow s_{i+1}) \in Transisions_{P_1}$, $(t_i : m_i \rightarrow t_{i+1}) \in Transistions_{P_2}$ and $Polarity(P_1, m_i) \neq Polarity(P_2, m_i)$

Informally this means that the two protocols will start at their respective initial states (the first collaboration state), and will then advance step by step, according to the same sequence of messages $(m_1, m_2, \ldots)$. The resulting sequence of states that each protocol goes through is called a *collaboration history*.

Let $Collabs(P_1, P_2)$ be the set of all possible collaboration histories between P_1 and P_2. $Collabs(P_1, P_2)$ give all possible "execution traces" between P_1 and P_2.

Protocols P_1 and P_2 have no *unspecified receptions* [3] if and only if for every pair $\langle s_i, t_i \rangle$ that is part of a certain collaboration history $\alpha \in Collabs(P_1, P_2)$, the following two holds:

- if $(s_i : m_i \rightarrow s_{i+1}) \in Transisitions_{P_1}$ and $Polarity(P_1, m_i) = -$ then $(t_i : m_i \rightarrow t_{i+1}) \in Transitions_{P_2}$ and $Polarity(P_2, m_i) = +$
- if $(t_i : m_i \rightarrow t_{i+1}) \in Transisitions_{P_2}$ and $Polarity(P_2, m_i) = -$ then $(s_i : m_i \rightarrow s_{i+1}) \in Transitions_{P_1}$ and $Polarity(P_1, m_i) = +$

Informally, this means that P_1 and P_2 have no unspecified receptions if and only if, whenever a collaboration α can reach the point where P_1 (resp P_2) is in a state where it can send a message m, P_2 (resp P_1) will be in a state where it can receive that message, and hence there exists some collaboration history in which that message is exchanged at that point.

Protocols P_1 and P_2 are *deadlock free* [3] if and only if, for all finite sequences $\alpha \in Collabs(P_1, P_2)$, $\alpha = \alpha_1 \rightarrow_{m_1} \ldots \alpha_{n-1} \rightarrow_{m_{n-1}} \alpha_n$, where $\alpha_n = \langle s_n, t_n \rangle$, then either:

- s_n and t_n are final states of P_1 and P_2, respectively, or
- there exist $\alpha' \in Collabs(P_1, P_2)$ such that α is a *strict* prefix of α'.

That is, P_1 and P_2 are dead lock free if and only if the collaboration α ends with both protocols in final states, or the collaboration can continue.

Protocols P_1 and P_2 are *compatible* if and only if they have no unspecified receptions, and are deadlock free.

This definition of protocol compatibility requires that when one party can send a message m, the other party must be willing to receive that message. However, the protocols are still compatible even then one party can receive a message m, yet the other party cannot send that message. This asymmetry reflects traditional programming environment, where the sender protocol decides what it wants to send independently of the recipient, and where the recipient has no control over what the sender sends.

Let $s \in States_{P_1}$ and $t \in States_{P_2}$. By definition, $s \sim t$ if and only if there exist a collaboration history $\alpha = \ldots \alpha_i \ldots \in Collabs(P_1, P_2)$ where $\alpha_i = \langle s, t \rangle$.

Informally this means that $s \sim t$ if at some point during a collaboration history in $Collabs(P_1, P_2)$, P_1 was in state s and P_2 was in state t.

Since the relation $\sim$ captures exactly those states $s, t \in States_{P_1} \times States_{P_2}$ that can appear together in a collaboration state of a collaboration history in $Collabs(P_1, P_2)$ we can apply the criteria for compatibility shown above using this relation. In other words, we can reformulate the definitions of unspecified receptions and deadlock using only the $\sim$ relation.

Let $Equiv(P_1, P_2) \subset StatesP_1 \times States_{P_2}$ be the smallest set such that:

- $\langle init_{P_1}, init_{P_2} \rangle$ is in $Equiv(P_1, P_2)$ and
- if $\langle s, t \rangle$ is in $Equiv(P_1, P_2)$, $(s : m \rightarrow s') \in Transitions(P1)$, $(t : m \rightarrow t') \in Transitions(P_2)$, and $Polarity(P_1, m) \neq Polarity(P_2, m)$ then $\langle s', t' \rangle$ is in $Equiv(P_1, P_2)$.

Therefore we have $\langle s, t \rangle \in Equiv(P_1, P_2)$ if and only if $s \sim t$. So a simple algorithm for testing if two protocols are compatible is to compute the $\sim$ relation and then use it to test for possible unspecified receptions and deadlocks.Figure 2.3 shows an algorithm for computing the $Equiv$ set of two protocols P_1 and P_2 while figure 2.4 shows how to determine if two protocols have no undefined receptions.

Let P_1 and P_2 two service protocols.

- Let $Equiv \leftarrow \{\langle init_{P_1}, init_{P_2}\rangle\}$
- Let *continue* $\leftarrow$ **true**
- while *continue*;
  - Let *continue* $\leftarrow$ **false**
  - for $\langle s, t\rangle \in Equiv$
    - if $(s : m \rightarrow s') \in Transisions_{P_1}$ and $(t : m \rightarrow t') \in Transisions_{P_2}$ and $Polarity(P_1, m) \neq Polarity(P_2, m)$ then
      - Let $Equiv \leftarrow Equiv \cup \{\langle s, t\rangle\}$
      - Let *continue* $\leftarrow$ **true**
- Output $Equiv$ as the result

Figure 2.3: Algorithm to produce the $Equiv$ set of two protocols

Let P_1 and P_2 two service protocols, and $Equiv(P_1, P_2)$ the set of all their collaboration states.

- for $\langle s, t\rangle \in Equiv$
  - if $(s : m \rightarrow s') \in Transisions_{P_1}$
    - unless $((t : m \rightarrow t') \in Transisions_{P_2}$ and $Polarity(P_1, m) \neq Polarity(P_2, m))$ return false;
  - if $(t : m \rightarrow t') \in Transisions_{P_2}$
    - unless $((s : m \rightarrow s') \in Transisions_{P_1}$ and $Polarity(P_2, m) \neq Polarity(P_1, m))$ return false;
- return true;

Figure 2.4: Algorithm to tell if two protocols have no undefined receptions

Let P_1 and P_2 two service protocols, and $Equiv(P_1, P_2)$ the set of all their collaboration states.

- for $\langle s, t \rangle \in Equiv$
  - if $(s : m \rightarrow s') \in Transisions_{P_1}$
    - unless $((t : m \rightarrow t') \in Transisions_{P_2}$ and $Polarity(P_1, m) \neq Polarity(P_2, m))$ return false;
  - if $(t : m \rightarrow t') \in Transisions_{P_2}$
    - unless $((s : m \rightarrow s') \in Transisions_{P_1}$ and $Polarity(P_2, m) \neq Polarity(P_1, m))$ return false;
- return true;

Figure 2.5: Algorithm to tell if two protocols have no deadlocks

Chapter 3

Protocol Oriented Service Deployment Platform

Contents

3.1 Introduction

This chapter will introduce the main purpose of this work: developing a service deployment platform around the idea of service protocols. Like mentioned earlier, there are a lot of existing tools supporting service protocols, but must of them are *design time tools*, that operate at the early stage of the service development process. Such tools include:

Protocol Manipulation Tools These tool lets you *graphically* create and manage service protocols (e.g in the form of finite state machines), and

export and import the created model into various file formats (most of which being XML-based). This might also include more advance protocol analysis features, like the ability to compare protocols for compatibility or replaceability.

Service Code Generation Tools These tools aim at aiding the developer by automatically creating a code template out of the service protocol to be supported, in the same way that code generators can produce code templates from IDL or WSDL interface description.

Service Code Analysis Tools The purpose of these tools is to *statically* analyse service code, looking for protocol compliance errors.

Most existing tools do not operate when the service actually start executing, and interacting with clients.

The purpose of this work is to develop a *runtime* service protocol tool, that will bring the benefits of service protocol to the service execution. The following sections will explore this idea in more details.

3.2 Service Model

This section will introduce a model of service interactions that will serve as a starting point and base for our work.

3.2.1 Multiple interfaces per service

A service will often have to interact with many others in order fulfill his task. In our model, a service defines explicit interfaces through which it can be composed with other services. For instance, a book seller service, which displays a catalogue and allows customers to place orders, may include a "logging" interface and a "filter" interface. The logging interface can be "*connected*" to a matching interface in a home financial management service that logs financial records, writes checks, etc. The filter interface can be "*connected*" to another service which prunes the catalogue using criteria such as personal preference, supplier location, and price. Composing the home shopper, logging, and filter services by connecting their interfaces would yield a compound service.

When two services collaborate with each other via particular interfaces, each sends and receives messages according to the protocols given by the interface specification.

Figure 3.1 shows an example service which needs to interact with four other services and thus exposes four different interfaces.

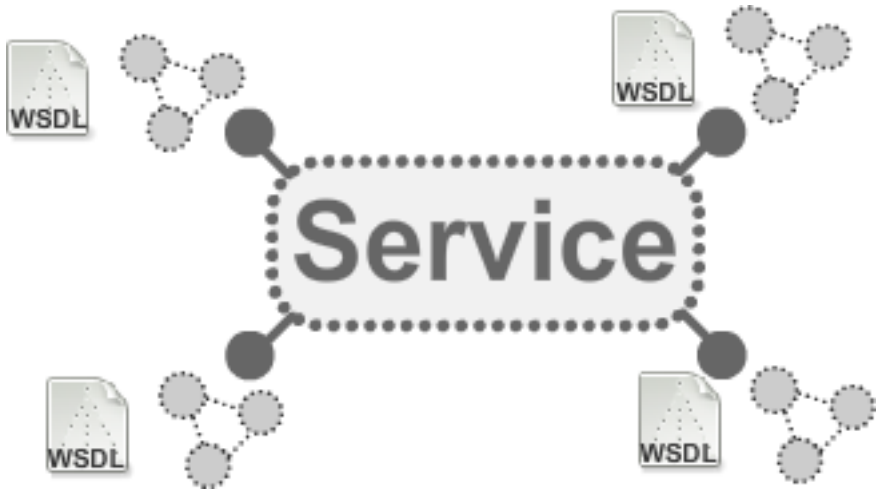

Figure 3.1: A service having multiple interfaces. Each interface having a corresponding WSDL specification as well as a protocol

When an interface of a service A is bound to an interface of service B, they are said to engage in a *collaboration*: messages sent through A's interface are received at B's interface and vice versa.

Note, however, that any particular collaboration is between exactly two parties. A collaboration between a service S and multiple other services is modeled by separate interfaces in S, one for each other party it is collaborating with. Figure 3.2 shows an example of service composition in the context of a travel agency service that needs to interact with the clients as well as the car renting service, room booking hotel service and the flight ticket reservation service.

3.2.2 Service Containers and Service Cores

From the above description of the service model, we can infer that the processing logic required to handle the protocols (maintain current state, and

Figure 3.2: A travel agency example. Each service can talk to multiple other services and has one interface per partner service

filter incoming and outgoing messages according to it) is common to all services. On the other hand, the actual number of interfaces, their respective protocols and more importantly the message handling implementation are service specific.

Thus, we can divide every service implementation into two parts:

Service Independent Code This part takes care of capturing the messages that come through a certain interface, checking them against the protocol of the interface, and advancing the current state of the protocol (or triggering an error if the message was unexpected).

Service Dependent Code This part specifies how much interfaces it needs (i.e how many other services it needs to interact with) the actual protocol for each service and a message handler for each possible expected message.

The service dependent code uses the service independent code as a "proxy" to send messages to other services and to receive messages from them. So that the service dependent code does not have to worry about maintaining the protocol current state or protection from unexpected messages.

From now on, the service independent code will be called the "*service core*" and the service dependent code will be called the "*service container*" (c.f. Figure 3.3).

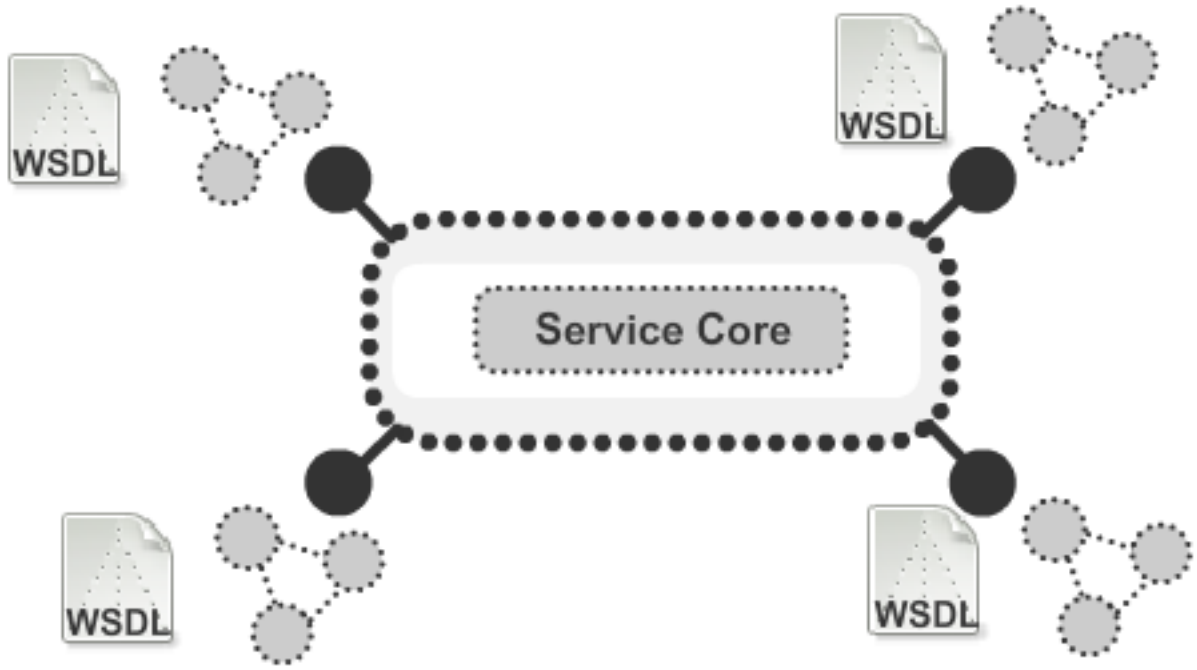

Figure 3.3: A service implementation is composed of two parts: the service independent code (the service container) and the service dependent code (the service core)

3.3 Protocol Oriented Service Deployment Platform

The purpose of this work is to bring the idea of service protocol to the service runtime. As such, we propose a "*protocol aware*" service platform that will host service cores developed around the idea of service protocols.

Specifically the role fulfilled by the platform will be:

Application Programming Interface : Offer the developer an easy way to declare how many interfaces are required as well as the service pro-

tocol associated with each one as well as a clear programming interface (API) to write the service implementation in terms of interfaces and service protocols.

Service Protocol Conformance Enforcement : At runtime, maintain the current state of each protocol, check input and output messages for protocol conformance, and notify the service core of any error.

Message Synchronization : Make sure the two sides of the interaction see the same sequence of messages (see below).

Interface Exposure : When a service is interacting with another "protocol-based" service (e.g when the other party is running on another instance of the same platform) the remote protocol is downloaded and checked for compatibility against the local one.

The following sections will detail each of the above features.

3.4 Message Synchronization

One important issue that arises when trying to implement service protocols is that of "*message synchronization*", which simply states that in order for the two services to communicate effectively they must "see the same sequence of messages". In other words, the sending operation must "appear" to be atomic with respect to both parties.

To understand why, consider the example of figure 3.4. The figure shows two services that are "in theory" compatible. Which means (according to the definitions given in the previous chapters that they have no unspecified receptions and that they are deadlock free. Yet, it is possible to "create an unspecified reception" if the following occurs: The first service (on the left side of figure 3.4 is initially in state 1, in which it can either accept a message A or send a message B, in this case, we assume the service decides to send B, and move to state to state 2 . In the meanwhile, and before the other service (before it receives the message B from the first one) can also either send a message A and advance to state 3 or receive message B and advance to state 2. Since the second service has not yet received the message B, it is still in state 1, so it decides to send A (see figure 3.5) As a result each of the services receives a message when it does not expect it.

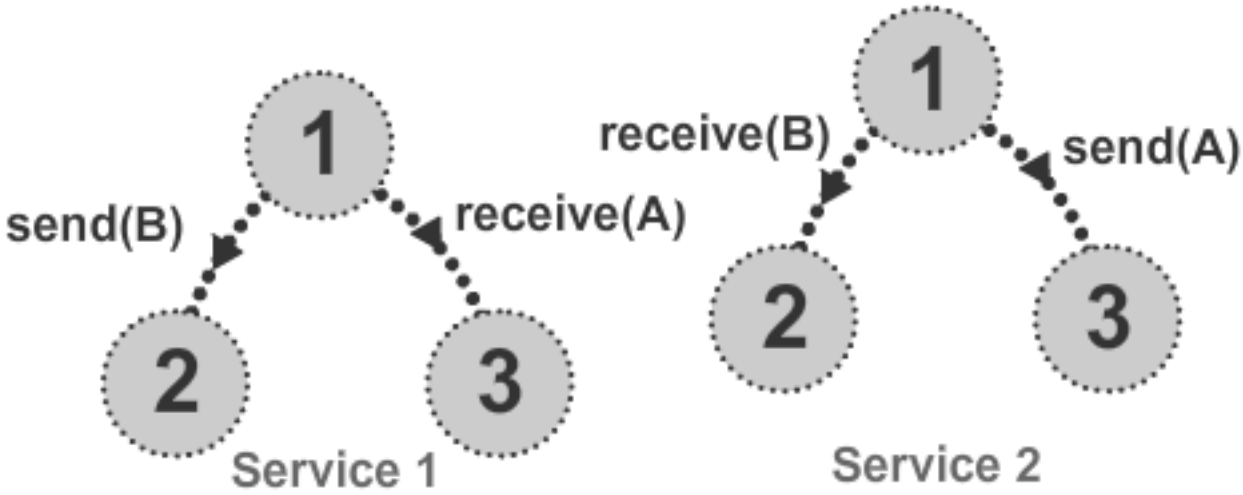

Figure 3.4: Two "theoretically compatible" services, but incompatible without message synchronization

This kind of "*protocol desynchornization*" is a result of the fact that service 1 "sees" the sending of his message as occurring first, before service 2 sent his message (and vise versae).

This kind of situation is pretty common in practice. For example, a numerical processing service may take a long time to process a particular request. During the calculations, it might be in a state where it can either *send the result* (if the calculation is finished) or *receive a cancellation request* to stop the processing. In a manner similar to the example of figure 3.4, without message synchronization it is easy to come across an unspecified reception even when dealing with a "compatible" protocol.

In order to provide message synchronization we propose employing the *totally ordered multicast* algorithm using *lamport logical clocks* [8]. Except that in this case there are only two parties involved in the communication. The algorithm will ensure that both parties will see the same order of messages.

On the other hand this means that some sent messages will need to be "aborted" or "rolled back". For example, reconsider the above example of a numerical processing service. If towards the end of the calculation process, the service decides that the operation is finished, and prepares to send the result, while at the same time the client decides that it wants to cancel the operation (no message has been exchanged yet), then no matter what

method we employ one of the two parties will have to give up on its decision and "roll-back" any action it has undertaken so far. Using lamport's clocks ensures that both protocols remain synchronized and agree on which message is going to be canceled and which one goes through.

The following sections will show how does this affect the application programming interface presented to the service developer.

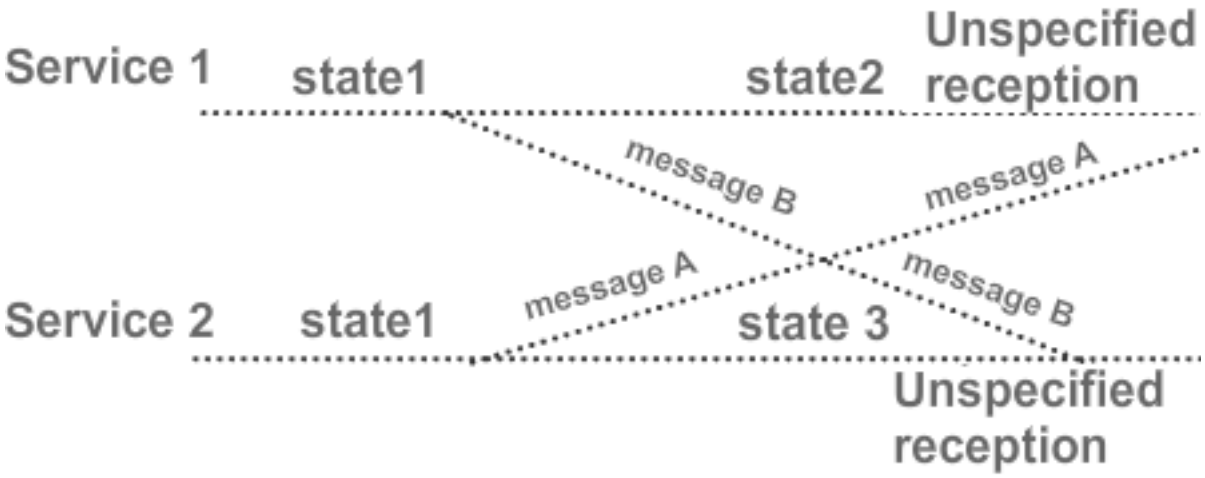

Figure 3.5: Timeline trace of a message exchange between the services of figure 3.4, that leads to unspecified receptions

3.5 Service Protocol Enforcement

The main purpose behind a runtime support for service protocols is to ensure that the communication occurs in conformance with the specified protocol.

This means that the platform will keep track of the current protocol state for each conversation the service is engaged in, so that:

- Before the service core can be notified of a message (and handle it) the message will have been checked against the current state of the protocol and make it advance to the next state. If the message was "unexpected" (according to the protocol), the error handling can be done by the platform itself without having to require the intervention of the service core.

- Before a message can actually be sent it is also check by the platform and will make the protocol state machine advance accordingly.

3.6 Application Programming Interface

The application programming interface is essentially the communication that is happening between the service core (that the service developer writes) and the service container (that the platform provides).

The message synchronization requirements seen in the previous section is one of the main influences of the application programming interface that will be presented to the developer. Indeed due to the fact that sending a message needs to be an atomic operation with respect to both parties we cannot simply provide a "*send_message(m)*" primitive that will immediately send the passed message (c.f. 3.4).

Instead the sending of a message must be a two step process:

1. The service core notifies the service container that he wishes to send a message M;
2. The service container takes care of ensuring that both parties agree on the same sequence of messages (by applying the totally ordered multicast algorithm[8], waits for acknowledgment messages, etc);
3. When the service container confirms that the message has been *delivered* to the other party, it notifies the service core which can then (and only then) assume that the send operation was effective.

When it comes to *receiving* messages, the service container can handle all the communication details and simply *deliver* the messages to the service core when they are ready to handled. This can be done either asynchronously in a "*pull mode*" where the service container keeps a queue of delivered messages and the service core has to interrogate the queue. Or in a "*push mode*" where the service container asynchronously "*interrupts*" the service core to let handle delivered messages.

3.7 Interface Exposure

Another very important application of service protocols is enhanced service discovery. Indeed, since we now have more information on a protocol behavior

it is possible to further filter search results based on the supported protocol.

In order to support this, the platform will provide a special operation, for each interface that the service exports, that will return the complete interface specification for the service: the list of supported operations and the associated protocol.

The information can be used by service registries (or service search engines) to further increase the likelihood of a successful search. It can also be used by service developer for creating client services (e.g by generation code templates from the interface description).

Chapter 4

Conclusion & Future Work

Most current efforts regarding SOA and Web services are mainly concerned with implementation and low level aspects (e.g mapping from WSDL to Java (and back), BPEL execution engines, etc). But there is a clear lacking support for higher level service analysis and management. The present work is oriented towards that aim.

Service protocols represent a firm foundation on which to base the next evolution of service composition. Their main benefit is that they provide extended interface description in a formal language. The formalisation permits to automate and extend the application field of *service composition*.

Many past works have concentrated on implementing the idea of interface protocols to component based models[12][4]. Those efforts contributed with models (e.g finite state machines), languages, and compatibility operations that can be carried on to the area of service orientation.

Existing approaches for service protocols explored many different ways to model service protocols, including petri nets[2] and state charts[7]. Many works seem to be concentrating on finite state machines and their variations[1]. In our work, we concentrated on finite state machine mainly for their simplicity and from the fact that surveys[5] of existing web services have shown that they are expressive enough to model most practical needs.

The main contribution of this work has been to show how we can leverage the benefits of service protocols at runtime. And how the common features can be abstracted away in a protocol oriented service deployment platform.

Among the possible future extensions and enhancements we propose:

- Explore the possibility of enhancing the model (e.g. adding the notion of time, where transitions could after a specified amount of time,

without any message being exchanged[5]);

- Service protocols do address any semantic issues, it would be interesting to bring some advancements made in the semantic web field to the area of service protocols;
- Explore the possible application of other protocol analysis operations (other than compatibility), like different levels of compatibility or replaceablility[5].

Bibliography

[1] Benatallah, B., Casati, F., and Toumani, F. Web service conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing 8*, 1 (2004), 46–54.

[2] Best, E., Devillers, R., and Koutny, M. Devillers and m.koutny: Petri nets, process algebras and concurrent programming languages. In *In: Advances in Petri Nets. Lectures on Petri Nets II: Applications, W. Reisig and G.Rozenberg (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1492* (1998), pp. 1–84.

[3] Brand, D., and Zafiropulo, P. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst. 30*, 2 (1983), 323–342.

[4] Canal, C., Fuentes, L., Pimentel, E., Troya, J. M., and Vallecillo, A. Adding roles to corba objects. *IEEE Trans. Softw. Eng. 29*, 3 (2003), 242–260.

[5] Casasti, F., Benatallah, B., Benatallah, B., Casati, F., Toumani, F., Toumani, F., Hamadi, R., and Hamadi, R. Conceptual modeling of web service conversations. Springer, pp. 449–467.

[6] Gouda, M. G., Manning, E. G., and Yu, Y. T. On the progress of communication between two finite state machines. *Inf. Control 63*, 3 (1986), 200–216.

[7] Harel, D., and Naamad, A. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol. 5*, 4 (1996), 293–333.

[8] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[9] Leymann, F. Web services: Distributed applications without limits. In *Datenbanksysteme in Büro, Technik und Wissenschaft - BTW2003* (February 2003), G. Weikum, H. Schöning, and E. Rahm, Eds., vol. 26 of *LNI*, GI, pp. 2–23.

[10] Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F., and Krämer, B. J. 05462 service-oriented computing: A research roadmap. In *Service Oriented Computing (SOC)* (2006), F. Cubera, B. J. Krämer, and M. P. Papazoglou, Eds., no. 05462 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. <http://drops.dagstuhl.de/opus/volltexte/2006/524> [date of citation: 2006-01-01].

[11] Vasters, C. Soa doesn't really exist, does it? http://vasters.com/clemensv/default,month,2005-05.aspx, 2005.

[12] Yellin, D. M., and Strom, R. E. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst. 19*, 2 (1997), 292–333.