SOA PATTERNS

Arnon Rotem-Gal-Oz

MEAP

MANNING

MEAP Edition
Manning Early Access Program
SOA Patterns version 14

Table of Contents

1

Solving SOA Pains with Patterns

How do you write a book on service-oriented architecture (SOA) patterns? As I pondered this question, it led me to many others. For instance, should I explain the context for SOA, or explain the background that is needed to understand what SOA is? Should I mention distributed systems? Should I discuss when an SOA is needed, and when it isn't? After much thought, it became apparent to me; a book on SOA patterns should be a practitioner's book. If you're faced with the challenge to design and build a SOA-based system, I've written this book for you.

You might *not* even agree with a SOA-based approach, but are perhaps forced into using it based on someone else's decision. Alternatively, you may think that SOA is the greatest thing since sliced bread. Either way, the fact that you're here, reading this, means you recognize that building an enterprise-class SOA-based system is challenging. There are indeed challenges, and they cut across many areas such as security, availability, service composition, reporting, business intelligence, performance, and so on.

To be clear, my goal here is not to lecture you on the merits of some wondrous solution set I've devised. True to the profession of the architect, my goal is to act as a mentor. I intend to provide you with the patterns that will help you make the right decisions for the particular challenges and requirements you'll face in **your** SOA projects, and enable you to succeed. However, before we begin our journey into the world of SOA patterns, there are three things we need to discuss first:

1. *What is software architecture?* Since the "A" in SOA stands for architecture, we need to define this clearly.
2. *What is a service-oriented architecture (SOA)?* This is an important question because SOA is an over-hyped and overloaded term. Therefore we want to make clear the definition that sets the foundation for this book.

3. *Explain the structure used to present each pattern in the book.* This will help them be more usable and readable to you as you move through the book.

1.1 Defining Software Architecture

There are many opinions as to what software architecture is. One of the more accepted ones by IEEE describes software architecture as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution [IEEE1471]. My definition agrees with this one, but is a bit more descriptive:

Definition: Architecture & Architecture Style.
Software architecture is the collection of fundamental decisions about a software product/solution designed to meet the project's quality attributes (i.e. architectural requirements). The architecture includes the main components, their main attributes, and their collaborations (i.e. interactions and behavior) to meet the quality attributes. Architecture can, and usually should, be expressed in several levels of abstraction, where the number of levels depends on the project's size and complexity.

Looking at this definition, we can draw some conclusions about software architecture:

- *Architecture occurs early*. It should represent the set of earliest design decisions which are both hardest to change and most critical to get right.
- *Architecture is an attribute of every system*. Whether or not its design was intentional, every system has architecture.
- *Architecture breaks a system into components and sets boundaries*. It doesn't need to describe all the components, but it usually deals with the major components of the solution, and their interfaces.
- *Architecture is about relationships and component interaction.* We are interested in the behaviors of the component as it can be discerned from other components interacting with it. Also, it doesn't have to describe the complete characteristics of components, but it mainly deals with their interfaces and other interactions.
- *Architecture explains the rationale behind the choices*. It is important to understand the reasoning as well as the implications of the decisions made in the architecture since their impact on the project is large. Also it can be beneficial to understand what alternatives were weighted and abandoned. This may be important for future reference, if and when things need to be reconsidered, and for anyone new to the project that needs to understand the situation.
- *There isn't a single structure that is the architecture*. There's a need to look at the architecture from different directions or viewpoints to fully understand it. One diagram, or even a handful, is not enough to be called architecture.

In order for a software system's architecture to be intentional, rather than accidental, it should be communicated. Architecture is communicated from multiple viewpoints to cater the needs of the different stakeholders. The Software Engineering Institute (SEI) defines an architectural style as a description of component types and their topology, together with a set of constraints on how they can be used.

1.2 Service Oriented Architecture

Arguably, the term SOA was first used in 1996 when Yeffim V. Natiz from Gartner defined it as "a style of multitier computing that helps organizations share logic and data among multiple applications and usage modes." Now, many years later, SOA is finally at the forefront of IT architectures and systems. However, on the uphill and rocky road to stardom, SOA has become a loaded term filled with misconceptions and hype. As in the game of "telephone," the definition of SOA has morphed as it was passed along in informal conversations. For the purpose of this book (and my view of SOA), we'll use the following definition:

Definition: Service Oriented Architecture

Service Oriented Architecture (SOA) is an architectural style for building systems based on interactions of loosely coupled, coarse–grained, and autonomous components called services. Each service exposes processes and behavior through contracts, which are composed of messages at discoverable addresses called endpoints. A service's behavior is governed by policies that are external to the service itself.

Before we dive deeper into this definition, lets take a brief look at some of the misconceptions and see why they're *not* SOA. Then we'll go back and expand some more on this definition, and its benefits both architecturally and business-wise.

1.2.1 What SOA Is, and Is Not

Many popular terms go through what Martin Fowler calls "semantic diffusion." For instance, as a term becomes more popular, people try to make them stick to whatever it is they're doing. This occurs either because they don't understand it precisely, or due to other, political, reasons. Additionally, the hype, or buzz, that a new term receives results in a lot of discussion around it. If the people discussing it don't understand it completely, the results are misconceptions and inaccurate descriptions.

For instance, in the late 1980's, object-oriented programming (OOP) was the hot new topic. As a result, developers referred to everything in their design, and their code, as objects simply because they wanted to say they were using object-oriented design and development techniques. The truth was, because the methodology was so new and the hype was so great, their descriptions were, in most cases, inaccurate. Therefore, it took several years for OOP to take root and for the development world to agree upon what it truly was.

One can argue that we're in the same stage with SOA; it has garnered many misconceptions and incomplete definitions. Table 1.1 details the most prevalent ones and explains why they are in fact - misconceptions:

Table 1.1- SOA, like many other popular terms, has created many misconceptions. As a term gets popular, people are more likely to brand whatever it is they are doing with it – whether it's an accurate description or not.

Popular SOA misconception	Why it's not SOA
SOA is a way to align IT and the business team	No, that's not true. Better IT and business alignment is something we want to achieve using SOA, but it isn't what SOA is. However, the loosely coupled systems that result from a good SOA solution enable the agility needed to truly align IT and the business team.
SOA is an application that has a "web service" interface	This is not necessarily true. To begin, we can implement SOA with other technologies. A nice example is the Open Services Gateway Initiative (OSGI), which defines a Java™-based service platform (see www.ogsi.org). Furthermore, exposing a method as a web-service can also be used to create procedural-like RPC, which is very far from SOA concepts and direction (see also NanoServices antipattern in Chapter 8)
SOA is a set of technologies (SOAP, REST, WS-I, and so on)	This is a general case of the previous misconception. Still, while some technologies are identified with SOA, or make a good fit when implementing them, SOA is an architectural approach. Remember, an SOA is technology-independent.
SOA is a reuse strategy	This not always true. Reuse certainly sounds like a tempting reason to use SOA – but the larger the granularity of a component the harder it is to reuse it. Nevertheless SOA will allow your "services" to evolve over time and adapt so that you don't need to start from scratch every time.
SOA is an off-the-shelf solution	SOA is not a product you can buy – it is a way to architect distributed systems. Perhaps you can resell the resulting service but that's only a

	convenient artifact of a good design.

After looking at these misconceptions, let's now re-examine the SOA definition provided above. SOA is an architectural style. This means that SOA defines components, relationships, and constraints about each component's usage and interactions. Following our definition above, the SOA style defines the following components: service, end point, message, contract, policy, and service consumer. SOA also defines certain interactions that the components can have. Figure 1.1 lists SOA's components and their relations:

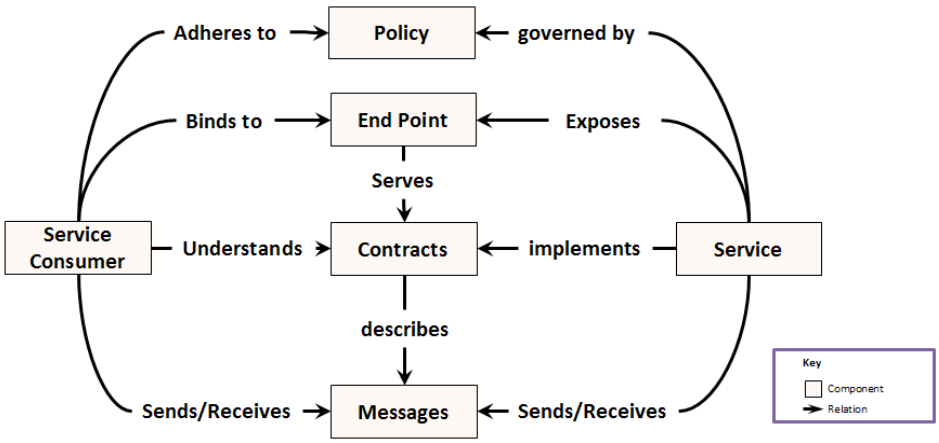


Figure 1.1 Apart from the obvious component (the service), SOA has several other components such as the contract that the service implements; End points where the service can be contacted; messages that are moved back and forth between the service and its consumers; policies that the service adheres to; and consumers that interact with the service.

Let's take a deeper look at each of the six components of SOA.

SERVICE

The central pillar of SOA is the *service*. Merriam Webster has eleven different definitions for the word service; the best is "a facility supplying some public demand." In my opinion, a service should provide a distinct business function, and should be a coarse-grained piece of logic. Additionally, a service should implement all of the functionality promised by the contracts it exposes. One of the characteristics of services is service autonomy, which means the service should be mainly self-sufficient.

CONTRACT

The collection of all the messages supported by the service is collectively known as the service's *contract*. The contract can be unilateral, meaning it provides a closed set of messages that flow in one direction. Alternatively, a contract might be bilateral, sending messages within a predefined group of components. A service's contract is analogous to the interface of an object in object-oriented design.

END-POINT

An *end-point* is a universal resource identifier (URI), such as an address or a specific place, where the service can be found. A specific contract can be exposed at a specific end-point.

MESSAGE

The unit of communication in SOA is the *message*. Messages can come in many different forms. For instance, they can be:

- HTTP GET messages; i.e. part of the Representational State Transfer (REST) style
- Simple Object Access Protocol (SOAP) messages
- Java Message Services (JMS) messages
- Simple Mail Transfer Protocol (SMTP) messages

The differentiator between a message and other forms of communication, such as a remote procedure call (RPC), is subtle. An RPC often requires the caller to have intimate knowledge of the other system's implementation details. With messaging, this is not the case. For instance, messages have both a header and a body (the payload). The header is usually more generic and can be understood by infrastructure and framework components without knowing implementation details. As a result, it reduces dependencies and coupling. The existence of the header allows for infrastructure components to route reply messages (e.g. the *Correlated Messages* pattern), or implement security transparently (see the *Firewall* pattern).

Messages are a very important part of SOA, and have been thoroughly covered by other books such as *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf [Hohpe03]. Nonetheless, this book also explores some messaging patterns where the SOA perspective enhances the more generic perspective used in [Hohpe03]. As an example, see request/reply and the correlated messages patterns.

POLICY

One important differentiator between SOA and object-oriented design (or even component-oriented design) is the existence of policy. Just as an interface or contract separates specification from implementation, policy separates dynamic specification from static/semantic specification. Policy represents the conditions for the semantic specification availability for service consumers. The unique aspects of policy are that it can be updated in run-time and that it is externalized from the business logic. The policy specifies dynamic

properties, such as security (encryption, authentication, authorization), auditing, service-level agreements (SLA), and so on.

SERVICE CONSUMER

A service is only meaningful if there exists another piece of software that uses it. Therefore, we define service consumers as software components that interact with a service via messaging. Consumers can be either client applications or other services; the only requirement is that they adhere to an SOA contract themselves.

1.2.2 SOA Architectural Benefits

By definition, SOA brings many architectural benefits to a distributed software system. For instance, many quality attributes are addressed, such as:

- Reusability — not in the sense of "write once integrate anywhere" but rather in the sense that "don't throw everything out when you need different functionality."
- Adaptability — isolating the internal structure of a service from the rest of the world lets you make changes more easily. You only need to adhere to the contracts you publish.
- Maintainability — services can be maintained by dedicated, smaller, teams, and can be tested this way as well. Robert. L. Glass once said, "software maintenance is a solution, not a problem" [Glass06]. SOA greatly helps make this a reality

These benefits exist because SOA removes the dependency issues related to point-to-point integration. Consider for example the situation in Figure 1.2.

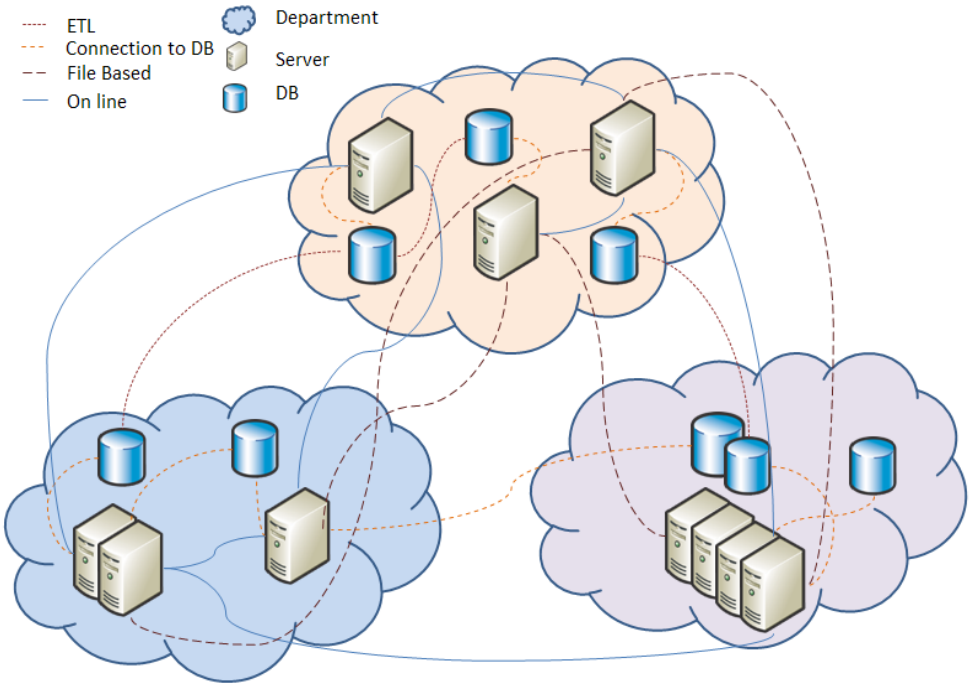


Figure 1.2 Typical Enterprise systems integration spaghetti. Each department builds its own systems. As people use the systems they find they need information from other systems and point-to-point integration emerges.

Many enterprises have grown isolated systems to solve particular business needs. These are sometimes referred to as *stovepipe systems*. As time passes and business needs change, there is often a need to share data between systems. Each time such a need is identified, a new relationship is formed between these systems. The result, as seen in Figure 1.2, is an integration mess that becomes very hard to maintain and evolve over time. The diagram shows four types of point-to-point integrations (Note that this is not an exhaustive list, there are additional relationships such as replication, message-based and others which are not expressed in this diagram):

- ETL (Extract, Transform Load) – a database-to-database relationship
- On-line – an application-to-application relationship based on HTTP
- File-based – an application-to-application relationship based on the file system
- Direct database connection – an application-to-database relationship

However, in a well-defined SOA, the interfaces are not designed to be point-to-point but are instead more generalized to serve many anonymous consumers. SOA eliminates this

spaghetti and introduces more disciplined communication. Fewer connectors means less maintenance, reduced assumptions. Fewer connectors also result in increased flexibility as shown in Figure 1.3.

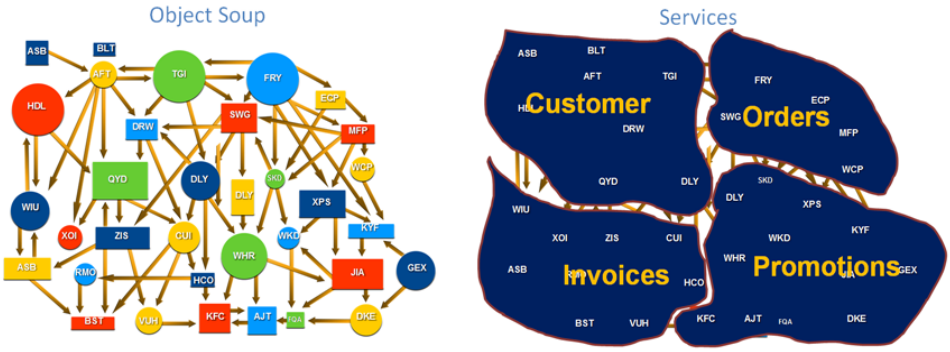


Figure 1.3 From Object Soup to well-formed Services; one of the ideas behind SOA is to set explicit boundaries between larger chunks of logic, where each chunk represents a high-cohesion business area. This is an improvement on the more traditional approach, which more often than not results in an unintelligible object soup

For enterprises that support a heterogeneous environment, with multiple operating systems (OS) and platforms, SOA provides standards-based contracts that are platform-independent. In fact, SOA enables transparent interoperability amongst services and applications across platforms.

Policy-based communications also greatly enhance the maintainability and adaptability of SOA-based solutions because several key aspects, like security and monitoring, are configurable. This moves some of the responsibility from the development team to the IT staff and makes life easier for both parties.

We can take all of these architectural benefits and translate them to business benefits, as discussed in the next section.

SOA an architectural style for the Enterprise

There are a lot of business-oriented aspects for SOA as well. SOA is described as a way to "increase the alignment of IT and the business." This allows for greater adaptability of IT to the changing business processes, and thus increases your business's agility. To avoid overloading the term SOA, we'd like to refer to these aspects of SOA as "SOA initiatives." Table 1.2 points out some of these business benefits:

Table 1.2 List of SOA technical benefits and the business benefits they provide.

SOA characteristic	Business benefit
Easier maintenance, replacement of components	Easier replacement of existing business components Better adaptability to accommodate changing business processes. Faster time to market for new business functionality
Standards-based service interfaces (contracts)	Reduced effort to connect new systems Easier partner integration Enable automation of business process
Service autonomy	Reduced downtime and lower operational costs.
Externalized policy	Ability to set service-level-agreements. Easier integration.

In general, it's better to take an incremental approach to adopting SOA. Your business cannot afford to halt and wait for the SOA initiative to finish, and you need to plan for SOA like highway intersections are planned; detours need to be created to enable business to continue while the new system is being developed.

Many SOA books cover the business aspects of the SOA initiatives – this book isn't one of them. This book's scope is on the software architectural aspects of SOA and technological implications of these aspects, not on the business analysis and related methods. One of the best ways to express these software architectural concerns and provide a better understanding of the architectural solutions is through the use of patterns (best practices) and anti-patterns (lessons learned, and mistakes to avoid)

1.3 Solving SOA challenges with patterns

With all the benefits mentioned above, why would anyone choose *not* to build with SOA? The truth is, building with SOA isn't easy. Even though SOA is designed to face the challenges of distributed systems design, there are still many issues you need to take care of and solve when you actually design viable solutions. One set of problems is the quality attributes--not inherently addressed by SOA--like availability, security, scalability, performance, and so on. Real projects have to deal with requirements like *five-nines availability* (99.999% uptime), which is no more than about five minutes of downtime per year.

Another set of problems has to do with the challenges of designing and building SOA. For instance, how do you gain a centralized view of business data in an architectural style that

encourages encapsulation and privacy? What does it mean to aggregate services? How do you tie your services to a user interface?

It would be nice if there were a few "best practices" already defined that could tell us how to cope with all of these issues. The truth is that there are no "silver bullets" in software design and development. Every system has its own set of prerequisites, hidden costs, one-off requirements, and special case exceptions. This is exactly why the use of patterns is so appealing as a medium to convey solutions. Patterns aren't defined to be a perfect solution. Instead, they give context where the solution works. To achieve this, patterns describe both the solution *and* the problem they solve, and caveats associated with that solution.

The following section explains the pattern structure used in this book and demonstrates how to apply the patterns to your own set of design challenges.

1.3.1 Patterns structure

Patterns in this book mostly take after what is called the *Alexandrian form* [Alexander77], which is named after the style Christopher Alexander used in his book, *A Pattern Language*. With this form, pattern descriptions are narrative with a few headings for readability, and serve as a vocabulary for both designers and architects. To start, each pattern has a descriptive name that is easy to remember and recall when applicable. The name is then followed by a short narrative passage to introduce the problem, which is the first heading. The other headings in the patterns description are Solution, Technology Mapping, and Quality Attributes. Let's examine the pattern form, and each heading, in more detail now.

PROBLEM

The problem section, as its name implies, details the problem the pattern aims to solve. It usually begins with a problem statement in bold that summarizes the essence of the problem. More complex problems have an additional passage, prior to the problem statement, that details the problem's context. For instance, some patterns contain an example to help illustrate the problem.

Following the problem statement, this section often continues with a discussion on other related. For example, there may be a discussion on alternative solutions and why they fail.

SOLUTION

The solution begins with a solution statement (again in bold face) that summarizes the essence of the solution. A diagram, which serves as a visual representation of the solution's components and their relationships, follows the solution statement.

The same diagram conventions are used for all the patterns with different visualizations for the SOA components, and other neutral players (see Figure 1.1). This includes component relationships, other pattern components, attributes, and the functionality of the pattern's components. Take a look at Figure 1.4 as an example.

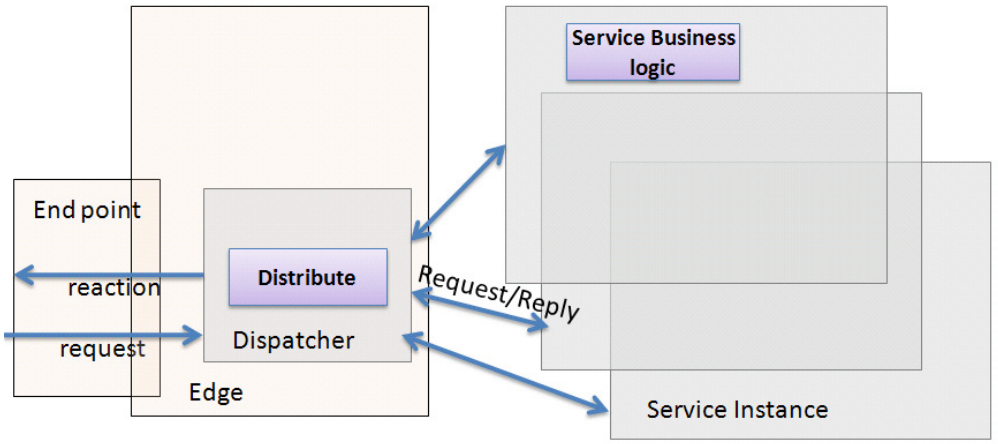


Figure 1.4 Sample pattern diagram: the *Service Instance* pattern. *Endpoint* and *Edge* are two neutral components (not part of the pattern).

Without getting into the details of the roles of the different components, looking in this diagram we can see that the Edge and Endpoints are neutral components that are not a part of the pattern. The Dispatcher and the Service Instance are two components that are a part of the pattern. Each of the pattern's parts has one or more roles and attributes. In this case we can see the Dispatcher is responsible for Distribution (of messages) and that the Service Instance is responsible for (running) the Service Business logic). *Dispatcher* and *Service Instance* are part of the pattern, while the purplish rectangles designate roles or attributes of the pattern's components (for instance, the Dispatcher distributes messages). The arrows are used to show interactions and relationships. For instance, requests and replies are passed back and forth between *Dispatcher* and *Service Instance*.

The pattern description then continues with more details regarding the solution, such as how the solution addresses outside forces, and so on. There may be a discussion on the implications or consequences of applying the pattern as well as the relationship to other patterns and examples.

TECHNOLOGY MAPPING

This section of the pattern description deals with technology implications. Although a system's architecture can be technology independent, a set of technologies must be chosen to actually build the system. Therefore, as a practicing architect, you often need to map parts of the architecture to specific technologies. For SOA, there are many relevant technologies, such as the WS-* protocol stack, REST-based web services, dedicated products, EDBs, and many others. The technology mapping section of each pattern talks

about the relevant technologies that can be used to implement the pattern or where the pattern is implemented.

QUALITY ATTRIBUTES

The final section of the pattern description has to do with identifying applicable patterns for your solution. In Figure 1.5, we see the various inputs the architect can use before a solution is designed. If patterns are the solutions, then quality attributes are the requirements. The quality attributes section of each pattern talks about the architectural benefits of the pattern, and provides sample scenarios that can be used to identify the pattern as relevant.

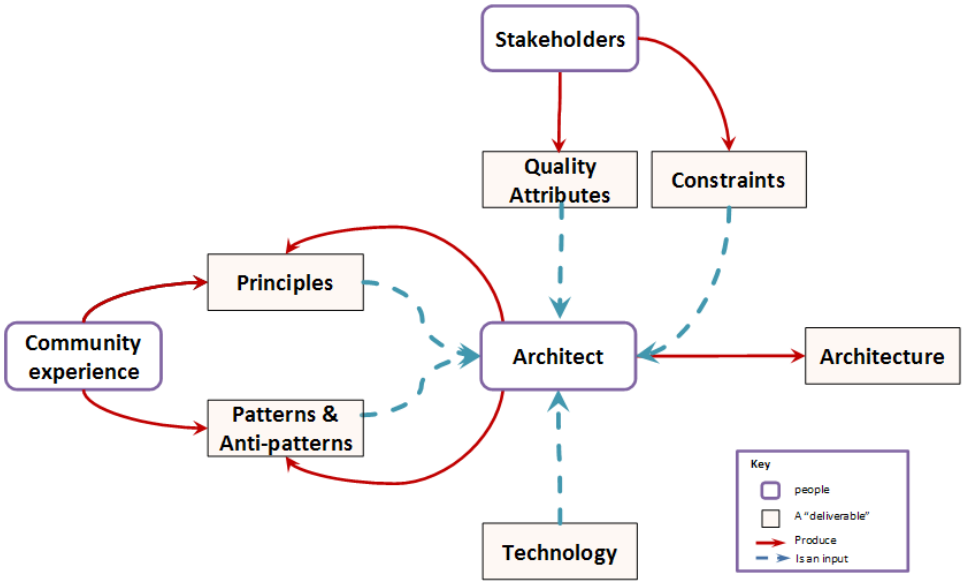


Figure 1.5 The Architect uses various inputs to design the architecture.

First and foremost, you work with the constraints and requirements gathered from the stakeholders. These include, requirements for performance, security, scalability, interoperability, and so on. You can augment these inputs by drawing on personal and community experience to add principles, patterns, and anti-patterns. There are also the possibilities and constraints imposed by available technologies. Finally, you must analyze, prioritize, and balance all of these inputs to produce a final architecture to suit the problem.

1.3.2 From Isolated Patterns to a Pattern Language

Each pattern on its own provides some useful information and describes a good practice. As mentioned, patterns have relationships to other patterns. For instance, sometimes another pattern is an alternative, and sometimes patterns can complement one another. There is usually value in documenting these relationships, and this structural organization is called a "pattern language".

Evolving patterns into a pattern language, which shows the patterns' relationships, helps increase the ability to recognize related problems, and allows the architect to navigate the patterns in a logical way. In a sense you can think of a pattern language as a logical and intuitive "mind map" of the patterns that lets you take different paths through the design process. As a result, patterns often open your mind to the bigger-picture problems that need to be solved, and provide the needed visibility you may not have had before (see Figure 1.6).

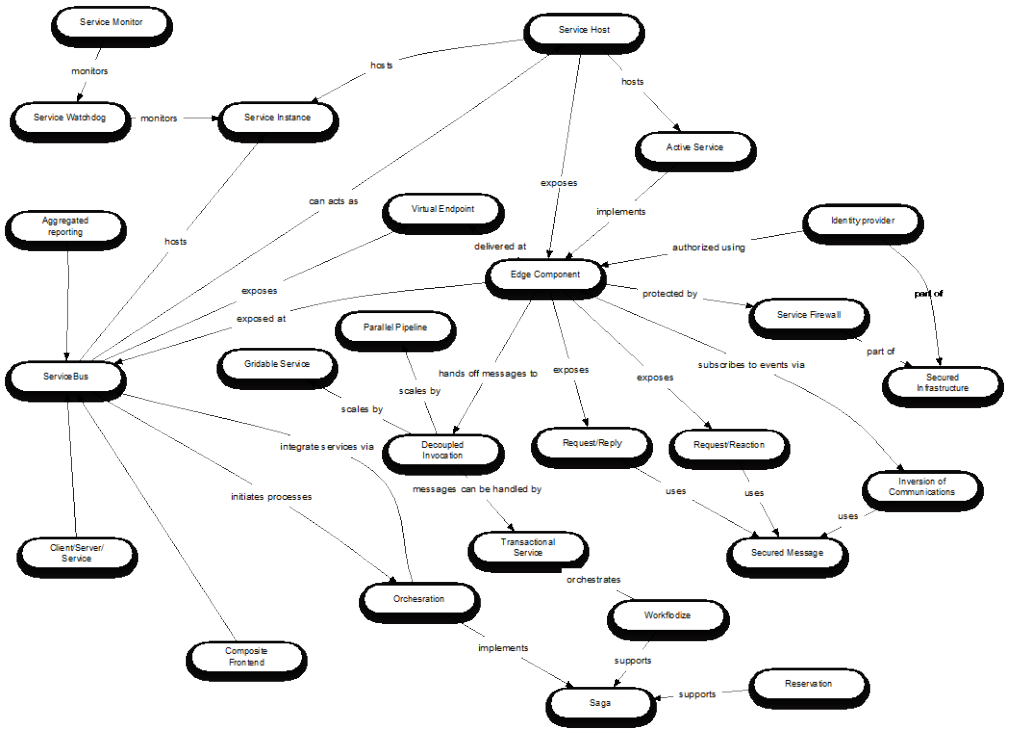


Figure 1.6 Like any good pattern language, the SOA patterns in this book build upon themselves to provide a "big-picture" solution.

Table 1.3 shows how the patterns in this book are categorized, and the chapter where they're discussed. Note that as we progress from chapter to chapter, we move outward. The first 2 pattern chapters mostly deal with the internal structure of the services. Chapter 4 focuses on the service interface, chapters 5 and 6 focus on the service consumer and its interaction with the service and chapter 7 on the SOA as whole.

Table 1.3 – List of pattern categories and their mapping to chapters

Category	Sub-category	Description	Chapter
Service Structure	Basic	Common service building blocks	2
	Performance, Availability & scalability	Patterns to solve scalability ,availability and performance challenges	3
	Security & management	Patterns for securing and managing services	4
Service Consumer Interaction	UI interaction	Interaction patterns when the consumer is a user client	5
	Service Interaction	Interaction patterns when the consumers are other services	6
Composition Patterns		Making services work together and share information	7

When you encounter a problem in your SOA implementation, you can use both the pattern diagram and the pattern categories in Table 1.3 as roadmaps to help you locate the patterns that can help you solve the pain. Using the patterns diagram you can also find related patterns to create more complete solutions.

1.4 Summary

In Chapter 1 we laid the foundation needed to understand the proposed SOA patterns and their overall context. We began with a definition of SOA and patterns in general, and how patterns can be used to provide solutions to SOA challenges. The definition was followed by a short discussion on the technical and business benefits of SOA. The second part of this chapter explains what patterns are, each pattern's structure, and provides a map for locating the patterns discussed in the rest of the book.

This chapter covered a lot of issues very briefly. The goal was to create a common vocabulary for our discussion on SOA patterns. If you're interested in learning more on some of the issues discussed in this chapter, look at one or more of the resources listed in the further reading section.

Chapter two is our first pattern chapter, in which we'll take a look at some of the basic patterns used to build services.

1.5 Further Reading

Table 1.4 resources for further reading on topics covered in this chapter.

Area	Resource name	Why
Distributed Systems	IT Architecture & Middleware: Strategies for building large and integrated systems – Chris Britton	Provide a good look a the history of distributed systems and the inherent difficulties that they inflict. It is a very thorough book, the only problem is that it ends just before the SOA era.
Fallacies of Distributed Computing	"Fallacies of Distributed Computing Explained" (http://www.rgoarchitects.com/Files/fallacies.pdf)	SOA is an architectural style for distributed systems. Most other styles don't have a "distributed mindshare" and thus, unlike SOA, they disagree with the fallacies. The link is a paper I wrote which explains how the fallacies are still relevant today.
SOA	Enterprise SOA: Service Oriented Architecture Best Practices - Dirk Krafzig et. al.	One of the best books on SOA. Provides a very good introduction on the subject
SOA	Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology - Eric A. Marks, Michael Bell	Takes a look at the business perspectives of SOA and provides a completely different (and complementary) angle at SOA (vs. this book)

1.6 Bibliography

- SSA *Research Note* SPA-401-068, 12 April 1996, "'Service Oriented' Architectures, Part 1", Gartner
- **[Alexander77]** Alexander, Christopher, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977
- **Arora Sandeep** EAI, BPM and SOA [Online]. - 9 December 2005. - http://www.soainstitute.org/articles/article/article/eai-bpm-and-soa/news-browse/1.html
- **Bass Len, Clements Paul and Kazman Rick** Software Architecture in Practice, 2nd edition [Book]. - [s.l.] : Addison Wesley Professional, 2003
- **Beer S.** The Heart of Enterprise [Book]. - Chichester : John Wiley & Sons, 1979
- **Britton Chris** IT Architectures and Middleware: strategies for building large, integrated systems [Book]. - [s.l.] : Addison-Wesley, 2000
- **Clements Paul, Kazman Rick and Klein Mark** Evaluating Software Architectures: Methods and Case Studies [Book]. - [s.l.] : Addison-Wesley Professional, 2002
- **Fielding Roy Thomas** Architectural Styles and the Design of Network-based Software Architectures [Online] // UNIVERSITY OF CALIFORNIA, IRVINE. - 2000. - http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- **Fowler Martin** Who needs an Architect [Article] // IEEE Software . - September/October 2003. - 11-13
- **Fowler Martin** Errant Architectures [Journal] // SD Magazine. - [s.l.] : CMP, April 2003. - 4 : 11. - 38-41
- **Garlan David and Shaw Mary** An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering, volume I. [Book]. - [s.l.] : World Scientific Publishing, 1993
- **[Glass06]** Glass Robert L. *software conflict 2.0: The art and science of software engineering,* published by Developer.* Books, March 2006
- **[Hohpe03]** Hohpe, Gregor, and Woolf, Bobby, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, October 2003
- **Hoogen Ingrid Van Den** Deutsch's Fallacies, 10 Years After [Journal]. - [s.l.] : Sys-Con Media, 8 Jan 2004. - 1 : 9
- **[IEEE1471]** ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems [Book]. - [s.l.] : IEEE, 2000
- **IEEE** IEEE 1061-1998 Standard for a Software Quality Metrics Methodology [Book]. - [s.l.] : IEEE, 1998. - ISBN 0-7381-1510-X
- **Rooten Luis d'Antin van** Mots d'Heures, Gousses, Rhames [Book]. - [s.l.] : Penguin,

1980. - ISBN 0-14005730-7

- **Rotem-Gal-Oz Arnon** RGO Architects [Online] // www.rgoarchitects.com. - 2005. - www.rgoarchitects.com/saf
- **Rotem-Gal-Oz Arnon** Dr. Dobb's Portal - Architecture & Design [Online] // www.ddj.com/dept/architect. - April 2006. - http://www.ddj.com/blog/architectblog/archives/2006/04/looking_the_sof.html?cid=GS_blog_arnon
- **Schulte Roy W. and Natis Yefim** V. SPA-401-068: "'Service Oriented' Architectures, Part 1 [Report]. - [s.l.] : Gartner

2

Foundation Structural Patterns

Congratulations, you are now in charge of building your first service – now what? The first thing to do, before getting into the advanced topics such as making your service secure and scalable, is to take care of the basic concerns. For example, where will you deploy your services? How do you ensure services' reliability? How do you enable anonymous access? And so on.

Chapter 2 deals with some foundation patterns, i.e. patterns that solve some of the more common issues related to all services. These are the patterns you are most likely to use, even if you have modest requirements for your services. In Chapter 1 we talked about SOA basics: creating autonomous components that publish and accept messages defined by contracts, delivered at an endpoint(s) and governed by policies to service consumers. Dealing with fundamental issues, the patterns in this chapter are relevant to implementing the services themselves (See Figure 2.1 below).

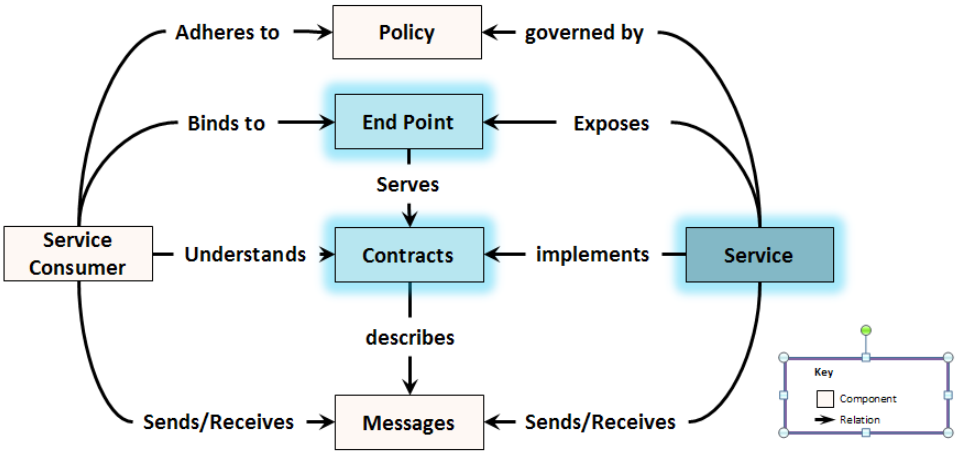


Figure 2.1 SOA defines six different components. Chapter 2 has patterns that deal with only one of them the services, which is of course the essence of SOA.

In chapter 2 we discuss five patterns. Table 2.1 below lists all of them along with the problems they address.

Table 2.1 – The list of patterns presented in this chapter.

Pattern name	Problem address
Service Host	How can you make your services adaptable to different configurations easily and save yourself the repetitive and mundane tasks of setting listeners, wiring components etc.
Active Service	How can I increase service autonomy and handle temporal concerns?
Transactional Service	How to handle messages reliably?
Workflodize	How to increase the service's adaptability to changing business processes?
Edge Component	How do we allow the service's business aspects, technological concerns, and other cross-cutting concerns evolve at their own pace, independently of one another?

Ok, enough introduction already, let's take a look at the first pattern which described the platform where our services would run.

2.1 Service Host

The first pattern we are going to talk about is one of the most basic patterns, if not *the* most basic one. The pattern is called *Service Host and it* deals with the environment where service instances run. Let's start by taking a look at the reason for why we need this pattern in the first place.

2.1.1 The Problem

Pick a service, any service (don't tell us what it is ☺). Wait, I think I see something… you have some code that sets up listeners for incoming messages or requests. You also have some code to wire-up components, and some more that initializes and activates that service. Further, you probably also have some code to configure your service Well, am I right? Chances are you have most of these pieces of code somewhere in your service.

The problem is you can end up having a lot of this code duplicated throughout the various services you've built, or will build. When building services, there are quite a few basic tasks that are repetitive and common.

> **There needs to be a way to easily configure services, and avoid duplicating the effort of mundane tasks such as setting listeners, and wiring components, for each service.**

The first option, one that's chosen all too often, is to rewrite the wiring and the rest of the repetitive code for each and every service. Obviously, this is not a good option; this results in wasted time, and can be error prone. The duplicated-effort problem is even worse when we consider maintaining a lot of similar code. For instance, if you make an enhancement or bug fix in this configuration code in one service, you'll need to copy to each other service that contains similar code. This is not an efficient use of your time, and requires an inordinate amount of testing to ensure you didn't miss anything.

A more reasonable solution is to create a library of common tasks and have each service work with a copy of it. A library helps – as code is written once, however you are still left with coding the wiring that is needed to utilize all of the library's functionality.

Another option is to use inheritance, i.e. create a base class that implements the common functionality, and have each service subclass it. However, inheritance can be problematic, especially if the service functionality doesn't fit within a single class. Additionally, inheritance will prevent us from using techniques like dependency injection to replace behavior/components Not to mention that this is the wrong use of inheritance, which should indicate an "is a" relationship. Nevertheless, inheritance comes close to solving the problem, as we only write the code once, and customization occurs where the services differ. If we want to get the same behavior without using inheritance, we can do that if we use a framework – a ServiceHost.

2.1.2 The Solution

Create a common ServiceHost component or framework that acts as a container for services. This container shall be configurable, and perform the wiring and setup of services.

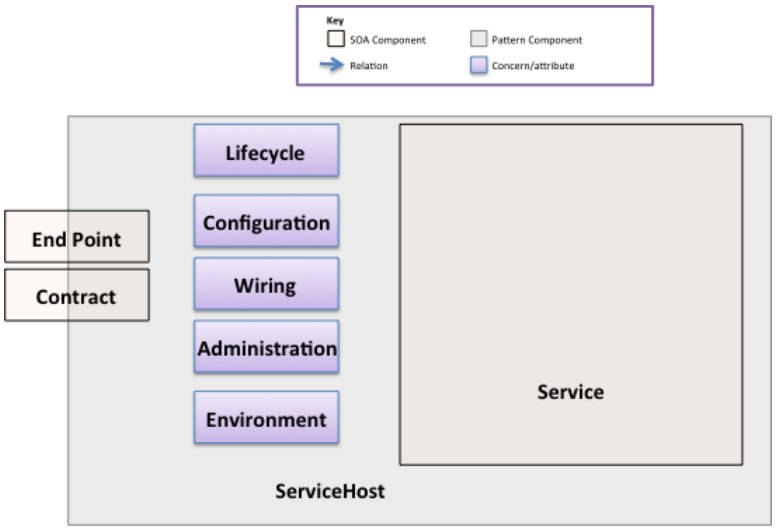


Figure 2.2 - ServiceHost is a container for a service and performs the wiring and configuration on the service's behalf

The ServiceHost is a framework or a complete component, which offers some or all of the following responsibilities:

- Lifecycle – Taking care of instantiating services, recycling services on fault, in-place upgrades etc.
- Configuration - Reading and applying configuration to hosted services. Configuration can include security, contract policies, ports etc.
- Wiring – run time setup of component wiring including, for example binding a listener on a service's endpoint
- Administration – An additional layer on top of lifecycle responsibility. Administration adds the capabilities to let an administration control the lifecycle of a hosted service and may also include monitoring capabilities
- Environment – providing auxiliary services like logging, cache, database libraries (ODBS/JDBC), scheduler etc.

The common denominator for these tasks is that they're all supporting capabilities that are needed by services. As we've seen in the problem introduction, you're likely to encounter them in more than one service.

Note that the ServiceHost is a framework , which means it contains functionality and data flow, and calls back into your code to extend the flow according to your service's needs. This callback principle is known as *Inversion of Control* (IOC), which is in wide use today in other object-oriented frameworks such as Spring, Hibernate, and Struts.

The ServiceHost pattern has several benefits when compared with the other options mentioned above. One benefit was already mentioned; as a framework the ServiceHost performs the work and only calls your code to fine-tune the behavior rather than leaving this orchestration to you. Another benefit is that it better addresses the *Open Closed Principle* (OCP). OCP states that a class should be open to extension but closed for modification, which is exactly what a framework gives you.

A single ServiceHost may host more than one service,, the number of services hosted depends on the scale of a deployed solution . For example, I've seen this pattern successfully applied on one case where a large solution had to be scaled down to run on a single computer. More often than not, the ServiceHost pattern is used to build services that span more than one computer, appearing as one aggregated service.

You can roll your own ServiceHost implementation, however, usually it is provided by technology vendors. We'll look into this in more detail in the technology mapping section below.

2.1.3 Technology Mapping

The ServiceHost is a fundamental SOA structural pattern and, as such, it's supported by most available technologies.

The most basic option is to roll your own ServiceHost. This is an option if you have modest or uncommon requirements. For instance on one system we created a ServiceHost since we needed stateful services on .NET platform and we didn't find something suitable from Microsoft. If you are implementing the ServiceHost pattern yourself, you should take a look at lightweight containers, such as Spring or PicoContainer, to help you out with wiring and instantiation. Nevertheless, in most cases it isn't the recommended way as there are plenty of options from technology vendors.

LIGHTWEIGHT CONTAINERS AND DEPENDENCY INJECTION

Spring and a few other frameworks are known as "light weight containers". They allow you to decrease coupling and increase the testability of your solutions. They perform this magic through the use of the Dependency Injection pattern, which is a non-SOA pattern. Dependency Injection occurs when a class lets a 3rd-party component, which acts as an "assembler", provide the entire implementation for the interfaces it depends upon. Using Dependency Injection, a class no longer has to depend on a specific implementation, but

rather it only depends on the interface or abstract class. This helps with testability as you can now supply stubs or mocks for the class to simulate its environment. It also helps flexibility as you can easily change the implementation of the dependencies without affecting your code, as long as they keep their contract.

Figure 2.3 below shows Microsoft's implementation of the ServiceHost pattern called AppFabric. It is a little hard to see the details in the image, but we can see is that AppaFabric (the servicehost) provides added value on top of hosting the services. You also get the means control the lifecycle of the hosted services, monitor them etc.

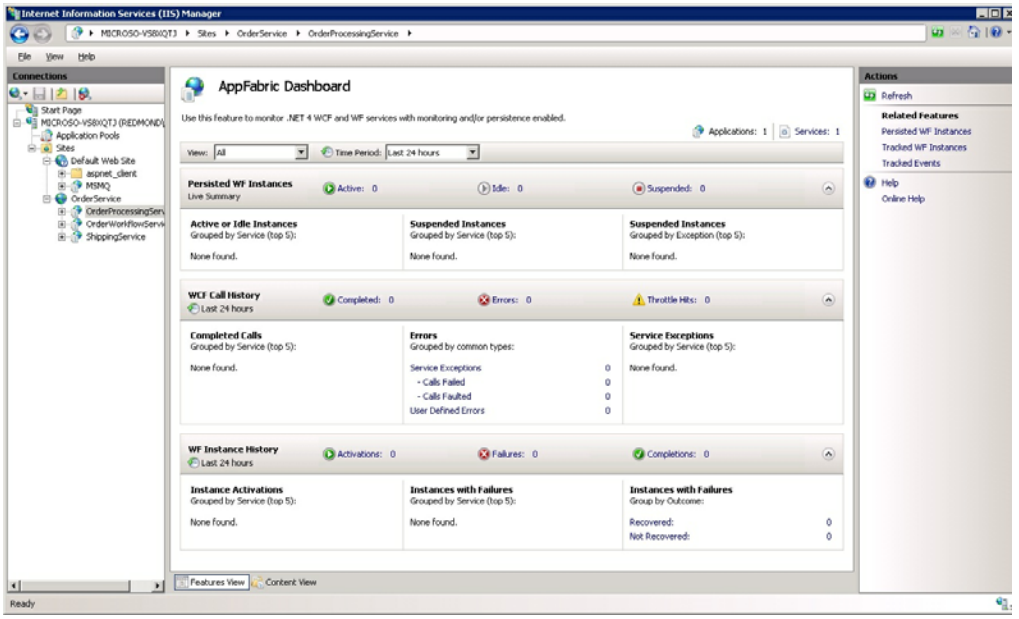


Figure 2.3 – A ServiceHost monitor dashboard. The ServiceHost has one service installed and it shows several statistics related to it like the number of calls, count of errors etc.

AppFabric is a relatively new addition to Microsoft's server stack. The Java world, on the other hand, has a relatively long tradition of application servers, most of them, like websphere or weblogic etc. can double as servicehosts. Most application servers support both JAX-WS (SOAP based web services) and JAX-RS (REST based web services). In addition to application servers, some Java Enterprise Service Busses (see also discussion on ServiceBus in chapter 7) provide service host capabilities. Figure 2.4 below shows the components of FuseESB, an open source, ESB based on apache ServiceMix. In the emphasized circle we can see the provisioning, deployment and admin capabilities (based on Apache Felix – an OSGi implementation)

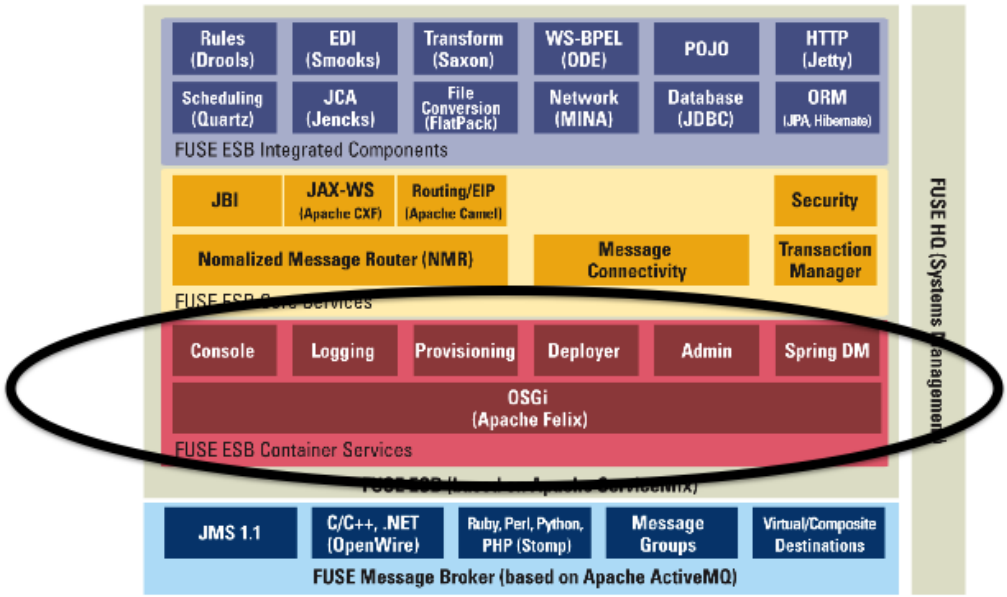


Figure 2.4 An Enterprise Service Bus with ServiceHost capabilities.

As we've seen, the ServiceHost pattern is basic` but effective, and is in wide use today. See the *Further Reading* section at the end of this chapter for links to resources that expand on the technologies mentioned in this section.

2.1.4 Quality Attribute Scenarios

The main reason to use the ServiceHost pattern is reusability. A nice side effect of reusability is the increased reliability you get as a result, since all of your services leverage a well-tested framework. The other quality attribute this pattern provides is portability, which is enhanced by the separation of concerns effect of the pattern, as was demonstrated in the scale-down example mentioned above. Another facet of portability comes from the ability to configure the service context in markup, which means you can deploy the same service code in different environments. Table 2.2 summarizes these attributes with two sample scenarios.

Table 2.2 – ServiceHost pattern quality attributes and scenarios.

Quality Attribute	Practical Advantage	Sample Scenario
Reusability	Reduce Development Time	During development, setup the environment for a new service within minutes.

Portability	Installation	During installation, switching from one environment to another should take little to no time.

A ServiceHost, as we've seen, is not unlike a webserver in many ways. It seems, then, that like web sites, services are passive by nature, i.e. a service will remain idle until a request arrives, at which time the service performs its work to generate a response. It turns out, however, that that's not always the best option , sometimes a service needs to be active rather than passive. Let's look at the Active Service pattern to learn why and how.

2.2 Active Service

We've already explained in Chapter 1 why it's important for services to be autonomous. To recap, it is because autonomy decreases coupling between services and spells greater flexibility for the overall solution. To be clear, this means that there are few dependencies between the services, as they only know each other by contract. It also means that the teams working on different services can be, in effect working independently. Each team focuses on its own service without interdependencies with other service implementations and their development teams. However, arguably the most valuable (as in "business value") definition is that the services are as self-sufficient as possible. Let's explain this with an example.

2.2.1 The Problem

Imagine a journal subscription agency like Ebsco or Blackwell, which needs to create a proposal for a client. The proposal service needs, among other things, to produce a *pro forma* invoice, which is a document that precedes the actual business transaction. In our sample scenario, to produce the pro forma, the service must provide both the discounts offered to the customer, and the discounts the business receives from its own vendors (the journals publishers). With this data, we can check if the proposal is profitable. Figure 2.5 shows a simple example for such a flow.

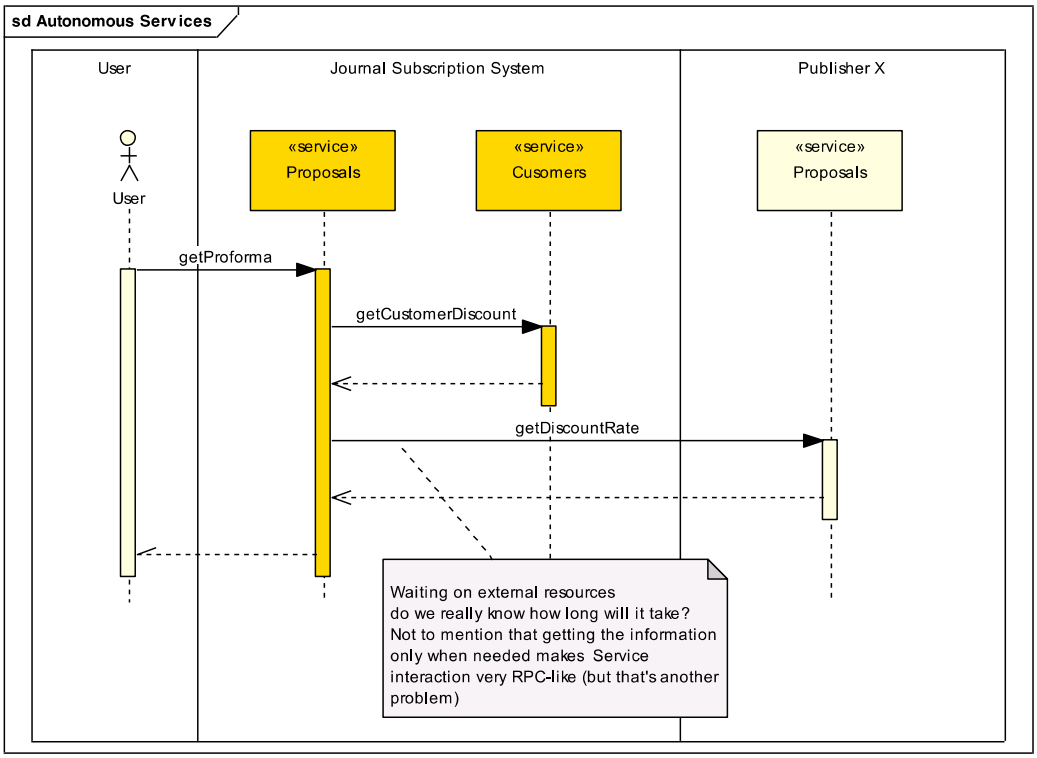


Figure 2.5. The Proposal service needs to get data from both internal and external services.

The proposal service needs to wait for the services it depends on to respond before it can send its own response. If either of the dependent services would fail, our Proposal service will be in effect unavailable. No amount of time, effort, or money spent in making the Proposal service resilient and fault tolerant will resolve such an outage. The reason is that the Proposal service is coupled too tightly to the other services. It might be acceptable to have this coupling between the proposal and the customer services as they are both internal and under our control. However the dependency on the external vendor's services is more risky – our Proposal service is not autonomous.

How can I increase service autonomy and handle temporal concerns?

As the example above demonstrates, a passive service that only reacts to requests is problematic. The service might not be able to fulfill its contract (or its SLA) if other services

don't behave as intended. Even when the external services are available a large load of requests can fail due to network congestion.

One option is for the service to cache previous results, but this is only a partial solution. For instance, it doesn't take care of data freshness so data in the cache can be too stale. Not to mention cache misses that will require external service calls anyway depending on the variety of requests this number may not be negligible.

Even if a cache solved our on-line requests problem, we still need to be able to solve other recurring or one-time events that are tied to time (which we refer to as temporal) - for instance, producing monthly bills or publishing stock figures, or any other recurring report.

A solution that can solve all that is to make our service do some work on its own accord – we need an Active Service.

2.2.2 The Solution

Make the Service an Active Service by implementing at least one active class, either on the edge, within the service, or both. Let the active class take care of temporal concerns and autonomy issues

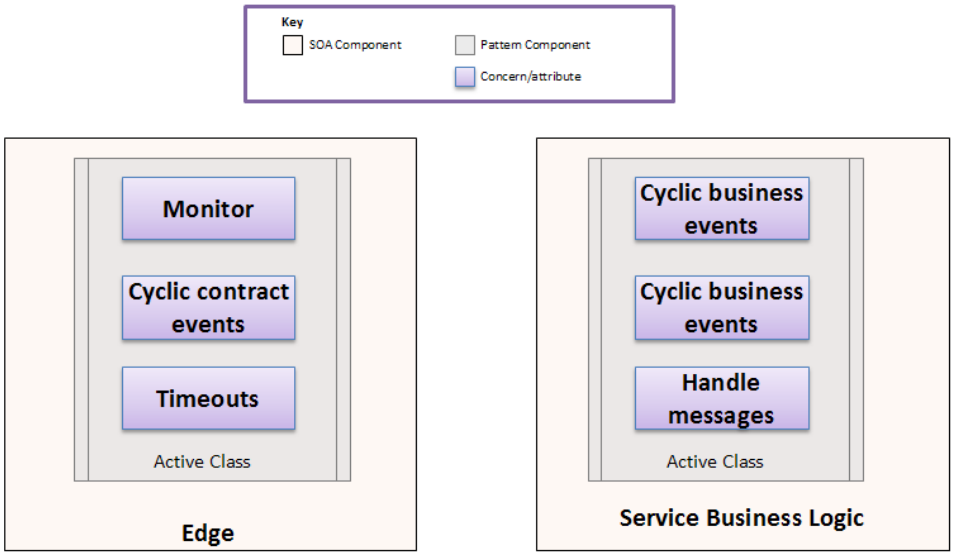


Figure 2.6 With Active Service, you add independent behavior to a service in its own thread of control. This can be used to handle cyclic events, timeouts, monitoring etc.

The Active Service pattern get its name from the object oriented concept called *Active Class. Active Classes,* as defined in the official UML specification, represent objects that may

execute their own behavior without requiring method invocation. The Active Service pattern means you implement the Active Class concept at the service level. As a result, the service creates worker threads to handle cyclic events such as monthly billing, report generation, and so on. A service can use this pattern, become active and monitor its own health, handle timeouts in addition to handling requests (see the *Service Watchdog* and *Decoupled Invocation* patterns in chapter 3).

How can an Active Service pattern help us solve the problems mentioned above? Well, Sometimes the best defense is no offense, i.e. instead of trying to solve the problem we can avoid the situation entirely. Instead of calling out to external services with each request to to the service, we can actively fetch data from other services to refresh the caches according to an independent schedule. This effectively decouples requests to the service from the connectivity and health of external services we depend on. Similarly, we can also proactively publish your own state changes (see *Inversion of Communication* pattern in chapter 5).

CACHING AND THE DENORMALIZATION PROBLEM

Those with a database background may read the suggestion to actively fetch and cache data from remote services and identify this as a potential data denormalization problem – what happens when the external data changes and the systems/services goes out of synch. Well, firstly, like any cache, the items in the cache should have a time-to-live or some other measure to assess their freshness. Secondly we should strive to make the data in the cache immutable e.g. by adding versioning so that a snapshot of the data is true for the time of the data. Lastly, we should strive to cache data that changes in low frequency if possible. In any event the owner of the data are the other services and we should keep that in mind when coding the service that uses the cached data.

A periodically scheduled thread (one that performs its work according to a timer) can take care of most of the other temporal events mentioned in the problem above, like producing a report. A thread in the Edge Component(s) (see section 2.5 below) is a good way to deal with contract-related temporal issues, such as timeouts events. A thread within the service can take care of purely business related concerns, such as sending monthly bill notices, or handling an incoming messages queue (see *Decoupled Invocation* pattern in chapter 3)

Let's look re-examine situation shown in Figure 2.5 and see how can we redesign it using the Active Service pattern. To recap, Figure 2.5 shows a flow for a proposal service that gets data from both an internal and an external service to produce a pro-forma invoice. Consider Figure 2.7, where the Proposal service actively goes to fetch data on a regular timely basis and caches the results. Thus, when a request to produce a pro-forma arrives, the Proposal service can immediately calculate the discount and return a reply. Using the Active Service

pattern, the Proposal service is decoupled in time from the services it depends upon to complete its work.

An alternate solution to this problem is to use the *Inversion of Communication* pattern (see chapter 5).

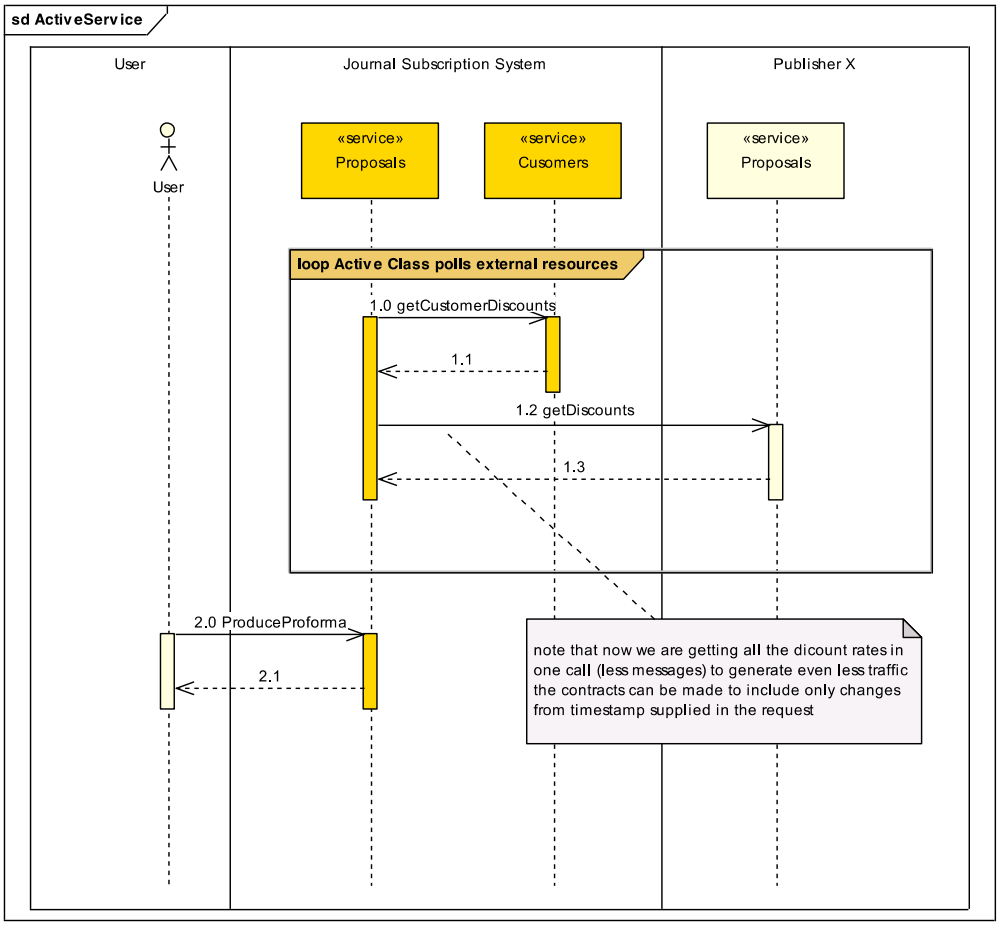


Figure 2. 7 The Proposal service actively polls the other services for the information it needs. Replies are sent immediately, and without dependency on the other services' availability.

Implementing, the Active Service pattern is rather simple, requiring little more than a child thread to handle asynchronous processing and data caching let's explain that further in the technology mapping section.

2.2.3 Technology Mapping

The idea behind the Active Service pattern is simply to have an active thread within the service, or in the Edge Component (see section 2.5 below) that will provide some specific functionality. As a result, the Active Service pattern relies on the threading capabilities of your implementation language or platform. It's important to decide exactly what you want to do with this thread in terms of external service call frequency, and data caching strategy= but these are general programming considerations and not in the scope of this book.

Let's take a look at few scenarios that consider the use of this pattern.

2.2.4 Quality Attribute Scenarios

Applied alone, the Active Service Pattern helps satisfy several quality attributes. However, the Active Service pattern is the prerequisite for many other patterns, such as Decoupled Invocation (see chapter 5) and Service Watchdog (see chapter 4), as mentioned above. These patterns further help to handle many quality attributes such as reliability, availability, and so on. Active Service also helps reduce overall latency because data is always available for the service to use in its response. As a result, application deadlines are met more often. Service availability is also increased, as services become more immune to failures of its services they depend upon.

Table 2.4 below list a few sample scenarios where the Active Server pattern can help.

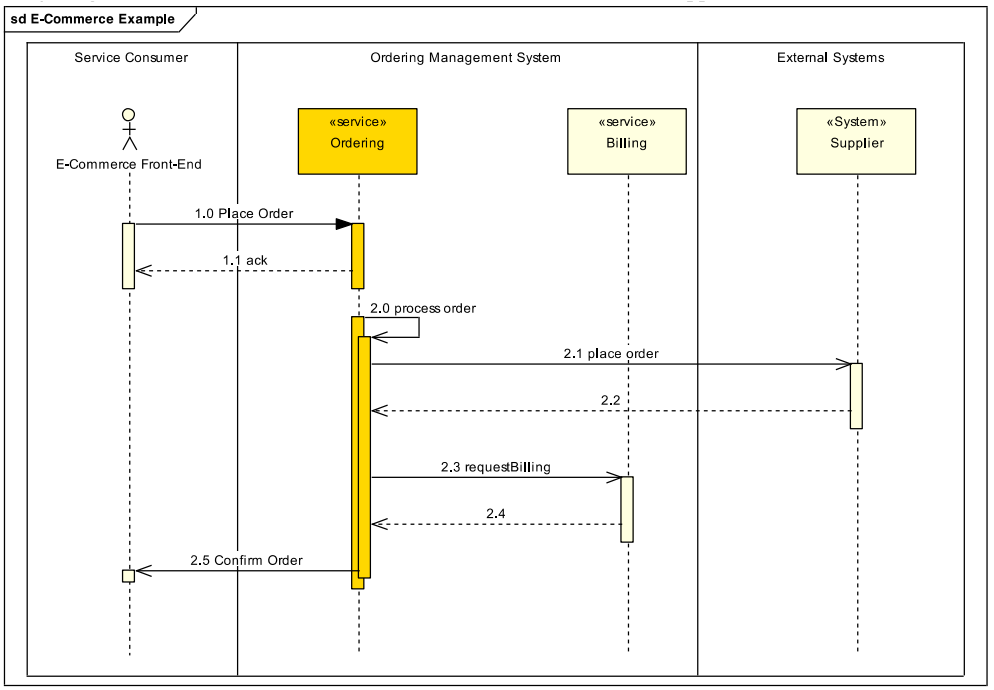
Table 2.3 The Active Service pattern includes many performance and quality-related benefits.s

Quality Attribute	Practical Advantage	Sample Scenario
Performance	Latency	Evaluating the profitability of an offer suffers no delay from external service calls
Performance	Deadline	Under load and normal conditions, the system can continue to update stock prices at regular intervals
Availability	Uptime	Even disconnected from the WAN, the service can still produce internal results

Moving forward, we need to consider how we handle messages once we get them either at the Edge Component or within the service. The *Transactional Service* pattern solves this problem, and it helps increase reliability.

2.3 Transactional Service

Discussing the Active Service pattern in the previous section we saw that a service may need to call other services to gather external data to perform its own responsibilities. Figure 2.8 below, illustrates such a scenario in an E-Commerce system. Here, a front-end component talks to an Ordering service (see the *Client-Server-Service* pattern in chapter 6 for more details on this type of configuration). The Ordering service registers the order request, sends the order to suppliers, and notifies a Billing service. When the order processing is complete, the service sends a confirmation to the E-Commerce front-end application (the service consumer in this example).



.Figure 2.8 The front-end sends an order to an ordering service which then orders the part from a supplier and asks a billing service to produce bill the customer.

The nominal scenario looks simple and clean, but what happens when or if something goes wrong? Let's take a look at this case now.

2.3.1 The problem

Now let's consider what might happen if the Ordering service crashes between acknowledging receipt of the order and processing it (for instance, between steps 1.1 and

2.0 in the Figure 2.8)? What would happen if the service fails just before requesting the Billing service to process the order, just before step 2.3.?

In both these cases the order is lost. Even worse, in the second scenario the system has already placed an order with the suppliers. The handling of messages in services is filled with situations just like this. Fortunately, things work most of the time, but as Murphy have it, our service is bound to fail eventually. Therefore, the question we must answer is:

How can a service handle requests in a reliable manner?

One solution to the reliability problem is to push the responsibility to the service consumer. If we consider the scenario above when the the service consumer doesn't get the order confirmation in step 2.5. It must assume that the order failed. However, this approach is not very robust, and it decreases the service's autonomy, as the service doesn't have any control over its consumers; they may or may not handle problems. Additionally, it only solves the problems that the service consumer is exposed to. What happens to the internal interactions the service makes? For instance, in the ordering scenario mentioned above, trouble will arise if the system fails after step 2.1, where am order is sent to the supplier. Clearly, this solution is not thorough.

Another option is to handle messages synchronously. However, synchronous operation can prove to be problematic in terms of performance, especially when the service needs to interact with external services, systems, or resources. As a result, each step in the entire process needs to complete serially before a reply can be sent. More importantly, this solution doesn't solve the problem entirely. For instance, if the service fails at any point, we have little visibility as to what problem actually occurred. The only thing we know for sure is that a message was lost.

A better solution is to have the service save its state in persistent storage, such as a database. This is a step in the right direction, but we need to ensure that the persistence mechanism is robust also. We need to know that the storage device (database, or otherwise) can track and record the process state if a failure occurs. To solve this issue, as well as the reliability problem in general, we need have defined the *Transactional* Service

2.3.2 The Solution

Apply the Transactional Service pattern to handle the entire message flow, from receiving a request message, to sending out a response, in a single transaction.

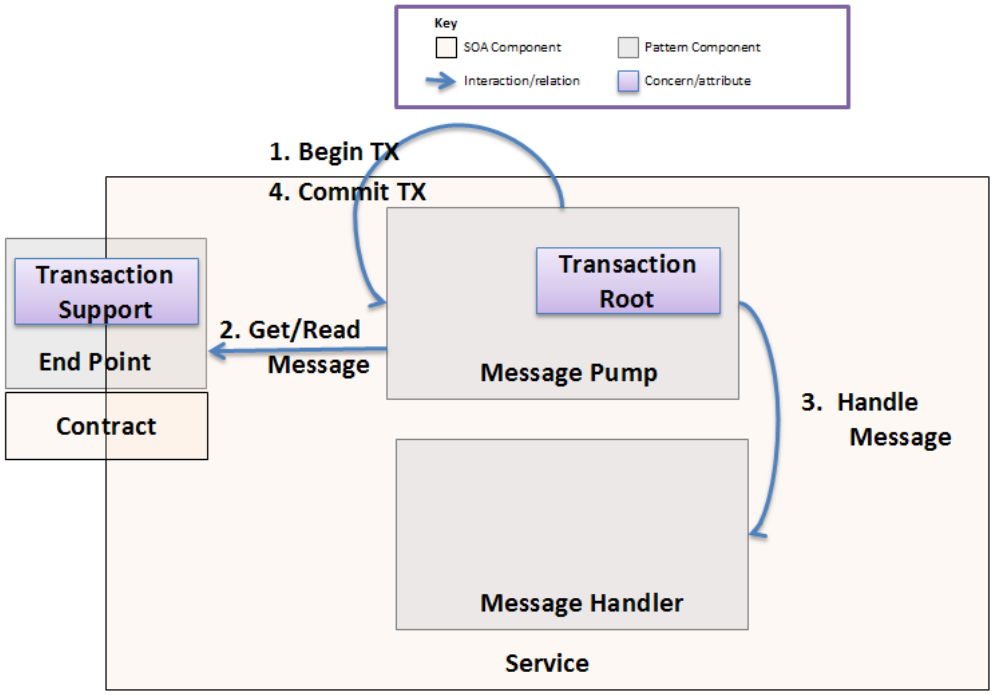


Figure 2.9 – The Transactional Service creates a transaction envelope: it opens a transaction, reads the request, handles the message, sends the response, and finally closes the transaction.

The main component of the Transactional Service pattern is the message pump, which listens on the endpoint or Edge for incoming messages. When a message arrives, the message pump begins a transaction, reads the message, passes it to other components to process the message, sends the appropriate response, and finally commits the transaction. You'll also need compensation logic in the case where the transaction aborts due to an error.

The advantage of a using a transactional programming model is that it ensures requests are processed completely, or not at all. This ensures data integrity is maintained, and that no requests are ever lost. For instance, if request processing fails at any step, all processing up until that point is rolled back, and the request is placed back into the incoming request queue (unless it is a problematic message which should be handled separately – see discussion on poison message below). Due to the properties of transaction, atomicity, consistency, isolation, durability--referred to as ACID properties as explained below--you're guaranteed that all of messages and related sub-operations are processed to completion.

ACID TRANSACTIONS

A transaction is a complete unit of work that demonstrates ACID properties, or qualities. An ACID transaction must be:

- **A**tomic – each step in a transaction occurs as one atomic unit. Either all the actions complete successfully, or none at all.
- **C**onsistent – Each resource is left in a consistent state, whether the transaction fails or succeeds.
- **I**solated – External observers (that don't participate in the transaction) never see the interim states. They see only the states before and *after* the transaction.
- **D**urable – Changes made in the transaction are saved in persistent storage so that they are available after a system restart.

In most cases, one tradeoff with the Transactional Service pattern is performance. Transaction processing can delay request processing due to the additionally preparation, the IO needed for durability, lock management, and additionally record keeping needed in case of a failure. One option when implementing the Transactional Service pattern is to use a transactional message transport for all the messages that flow between the services. This makes implementing the pattern much easier as you leverage the qualities built into such a message service.

Another option is to place request messages into a transactional resource, such as an enterprise queuing system, and then manually commit the transaction after a response is sent. In this case, note that, since the initial message handling is not transactional (it occurs before you place the request into the transactional queue) you need to be able to cope with a duplicate request arriving multiple times if the acknowledge message back to the consumer is lost (see the discussion on idempotent messages in chapter 6).

Figure 2.10 below shows a redesign of the example in Figure 2.8 using the Transactional Service pattern. To recap, the scenario illustrates an e-commerce front-end, which connects to an Ordering service. The Ordering service registers the order, sends the order out to suppliers, and notifies a Billing service. When each step is complete, it sends a confirmation message to the e-commerce front-end application.

Using the Transactional Service pattern, the actions taken by the ordering service itself (steps 2.0 to 2.5 in this diagram) occur within the same transaction. Therefore, if any step in the order process fails, each of the other steps already completed will be rolled back as though they never took place. A subtle issue here is what might happen if the Ordering service crashes somewhere between step 1.0 and 1.2.

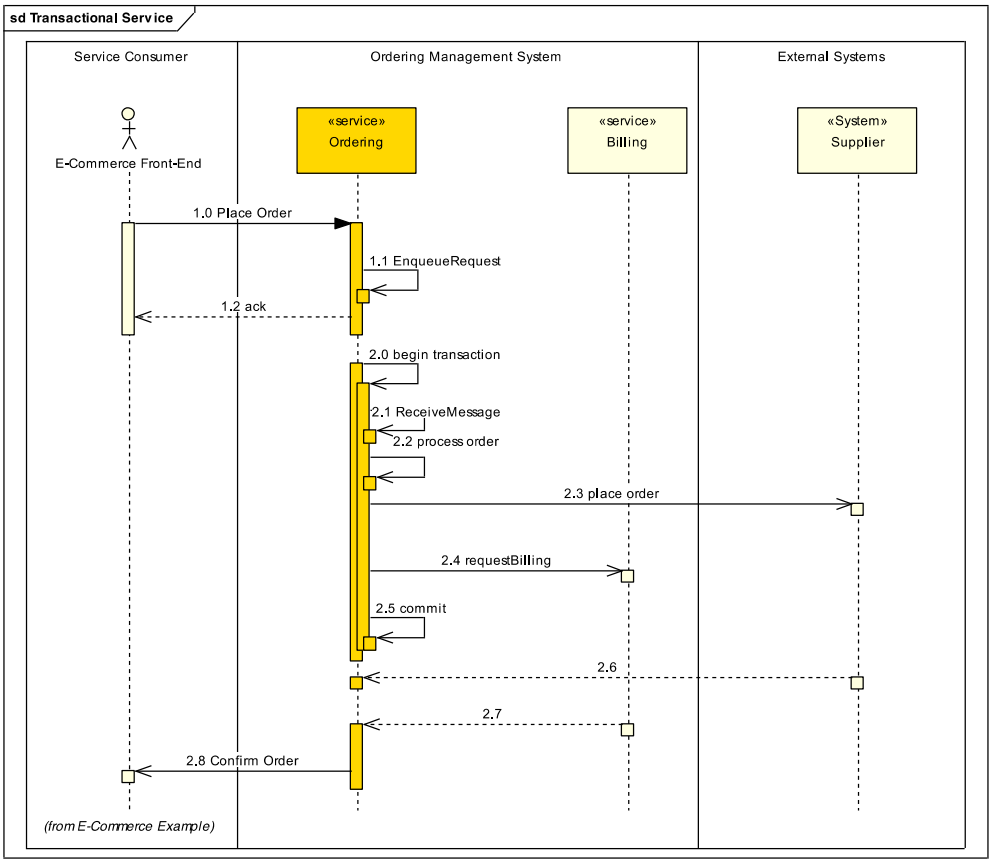


Figure 2.10 – The E-Commerce flow from Figure 2.6 redesigned to use the Transactional Service pattern.

Using a single transaction would work only if the Billing process produces an invoice *only*. It won't work if the Billing service also needs to process a credit card, which requires an additional confirmation to continue. When a single transaction isn't enough, the process needs to be broken into smaller transactions and the whole process becomes what's known as long-running operation (see the *Saga* pattern in chapter 5). Additionally, request processing may need to be broken into smaller transactions if the service itself is distributed across multiple computers, or even geographically.

There's an important difference in the transaction envelope when the transaction begins within the service (when a request is received), or at the sender when the request is first sent. Although transactions started outside of the service can help with reliability in terms of sender issues, it also increases coupling in the system. When you extend a transaction beyond the service boundary, holding internal resources for something beyond the service

trust boundary can be risky. We'll examine this in more detail in our discussion of the Transactional Integration anti-pattern (chapter 8).

Our next step is to look at what's needed to implement a transactional service.

2.3.3 Technology Mapping

Implementing the Transactional Service pattern can be easy if the message transport is transaction aware. Examples can be found in most enterprise service bus (ESB) software (such as, Websphere ESB, Apache ServiceMix and others), messaging-oriented middleware such as Microsoft Message Queue (built into .Net), any JMS implementation (such as WebSphere MQ or ActiveMQ), or even SQL Server's Service Broker. The process then is for the service to read a message from the ESB or messaging middleware, process it, send new messages to outgoing destination queues, and commit the transaction at the end to indicate success. If any of the individual components fails, the entire transaction is rolled back.

You may need to start a distributed transaction if, for example, you also perform database updates within the same unit of work. , A distributed transaction, sometimes referred to as a two-phase commit (2PC) transaction, coordinates more than one resource. For instance, it can coordinate the transaction engine within the queuing system with that in a database used to save any state changes after handling each message. In .NET 2.0 and later, you can open a TransactionScope object (defined in System.Transactions) to transparently move to a distributed transaction if needed. Similarly, Java code can use the transaction engine built into a Java EE-compliant application server, or other transaction service.

POISON MESSAGES

When we read a message in a transactional manner, we need to pay attention to identify and handle poison messages. A poison message is a message that is faulty in some way, and as a result makes that service crash or always abort the transaction when it is handled. Within a transaction, the problem is compounded because with each failure, the message is re-queued. Once the processing service recovers, it reads the same request again, fails, and repeats the cycle. Most enterprise messaging products automatically detect and discard poison messages (via what is called a dead-message queue) to help you avoid this scenario. You need to make sure this case is handled for you, or at least be aware of this and deal with it yourself.

A technology specification that may seem related is WS-ReliableMessaging. However, despite its name, the protocol is only concerned with delivering the message safely from point to point (effectively making it act like TCP for the HTTP protocol). There is no durability promise or any transactional trait imbued in the protocol. Many ESBs, which are transactional, also support WS-ReliableMessaging, making it a good choice in most situations. Other related protocols are WS-Coordination and its related specifications, WS-AtomicTransaction, and WS-BusinessActivity. We'll look into WS-BusinessActivity in more

detail when we discuss the Saga pattern in chapter 5. However, we'll avoid WS-AtomicTransaction--which defines a protocol to orchestrate a distributed transaction between services--because it introduces a lot of coupling between services (again, you can see Transactional Integration anti-pattern in chapter 8 for more details).

As usual, we'll end by looking at some of the motivations to use the transactional service pattern.

2.3.4 Quality Attribute Scenarios

The semantics that the Transactional Service pattern introduces can simplify both coding and testing. No longer do you need to write explicit error-handling code for each step of service processing. Most importantly, it greatly enhances the reliability and robustness of the service. The code becomes simpler, and you can focus on writing business logic, not error handing code. Here are two samples of success for the Transactions service pattern.

Table 2.4 The architectural scenarios that can make us consider using the Transactional Service pattern.

Quality Attribute	Practical Advantage	Sample Scenario
Reliability	Reduced data loss	A message acknowledged by the system will not be lost
Testability	Test coverage rate	For all critical reqs, Achieve 100%, test coverage

Another pattern that can reduce the amount of code hat needs to be written is the Workflodize pattern. Let's examine this pattern in detail now.

2.4 Workflodize

On one project, we had to build a sales support system for a mobile operator. It would probably not come as a surprise if I told you that the competition between mobile operators is *quite* fierce. The result of this competition includes much effort to define new usage plans and bundles to meet internal goals as well as customers' requirements. For the operator in question, new usage plans were created several times a week. Considerable time and effort was required to adjust the billing system to the new plans. However, marketing requirements often pushed the development teams into fire-fighting mode to implement the changes in record time.

Changing business needs is something that is common to many, if not all modern businesses. The degree of intensity may vary from system to system, but we've all experienced it at one point or other. Therefore we need to find a way to enable our services to efficiently cope with these changing processes.

2.4.1 The Problem

How can we increase the service's adaptability to changing business processes?

The most obvious option is just to wait for the change requests, and develop the code and update the services every time a requirement changes. This poses several problems; first, you need a full development cycle to make the change happen. Second, code changes require test efforts, which translates to even longer time to market. As an anecdotal evidence, in the project mentioned above , implementing changes to a plan, or adding a new plan, took three or more weeks – clearly too long to the business people involved.

Nevertheless, implementing the orchestration for the stable logic is still a daunting and error-prone task. There must be an acceptable solution.

2.4.2 The Solution

Introduce a workflow engine within the service to handle the volatile and changing processes and orchestrate the stable logic.

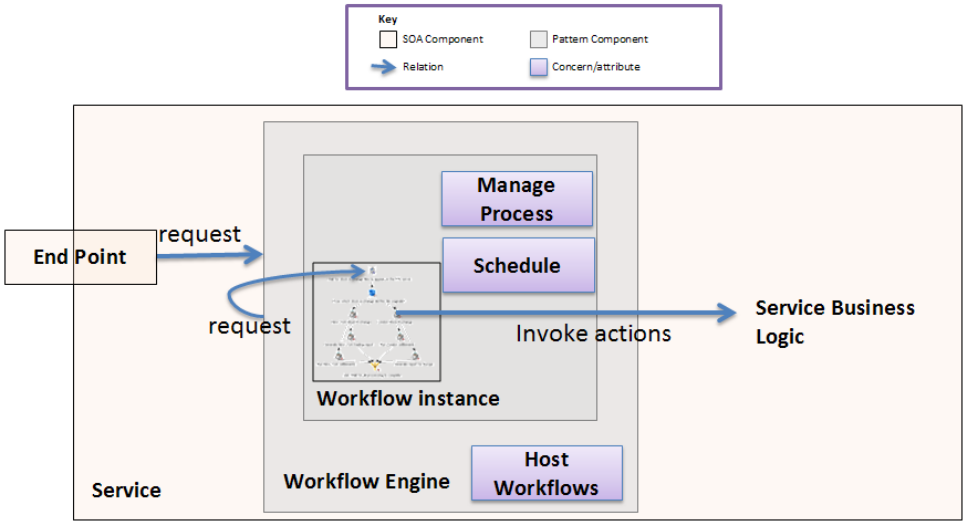


Figure 2.11 – The business process is made of the individual small building blocks that are relatively easy to rearrange. The workflow drives the business logic.

The Workflodize pattern, as depicted in figure 2.11, is based on adding a workflow engine to the service to drive business processes. The workflow engine hosts instances of Workflow. The nominal case is a workflow per request type. Workflows can also become quite complex;

for example, to handle long running processes with several entry points, where requests and responses arrive from external services.

The advantage of a workflow is that it gives you a tool that both makes you think in terms of building blocks (called activities), and it let you arrange and rearrange them into processes in a very flexible way. You model the processes as a flow of activities that occur as messages arrive. Since each activity can be tested individually, reusing them requires less testing overall. The flexibility for rearranging the activities means you can quickly respond to changing business needs, with less risk.

Given this flexibility, how does this impact the service's contract? Of course, this depends, but usually a change in internal implementation shouldn't ripple out and affect the contract. After all, the whole point of the contract is to shield server consumers from such changes. If we apply Liskov's Substitution principle to SOA (see the callout) there's no need to change the contract version if the overall behavior remains the same.

LISKOV'S SUBSTITUTION PRINCIPLE FOR SERVICES

Liskov's Substitution Principle, which is also known as *design-by-contract*, is an object-oriented principle that Barbara Liskov originally published, as follows: "If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T." [Wikipedia] In other words, this means that a subclass can be used in place of its parent class without breaking the behavior of any users of the base class. Applied to SOA, this means that when changing the internal behavior of a service, you don't need to create a new version of the contract. In other words, in order to keep the same contract version, the new version of the service should meet the expectations that consumers of the original service have come to expect.

Let's take another look at the scenario we discussed above and see how it looks with when we apply the workflodize and if it helps. To begin, we can use a workflow to route requests for new plans that don't require human intervention. For example, we can let the customer service department register the change in the CRM system, and then notify technicians to configure the network. Later, when the backend systems are ready, data can rerouted through them. The existing, stable, processing components represent reusable activities in the flow that all mobile usage plans leverage.

Adding a workflow in this scenario greatly enhances the business' ability to react and remain agile. When a competitor launches a new plan--which happens frequently in the mobile world--this mobile operator is able to react and launch a competing plan within a day or less. This is real and tangible business value.

The ability to handle long running processes mentioned above is another advantage to the Workflowdize service pattern. It can also be combined with other patterns. For example,

it's easy to add job scheduling (which most workflow engines support) to implement the Active Service pattern.

A pattern closely related to Workflodize is Orchestration (see chapter 7); both patterns use the same underlying technology: a workflow engine. Nevertheless, there are different architectural considerations that distinguish the two. For example, Workflodize is constrained to stay within the boundaries of a single service, while Orchestrated Choreography is used to coordinate multiple services.

2.4.3 Technology Mapping

The natural technology mapping for the Workflodize pattern is the use of a workflow engine. There are many workflow engines on the market, such as Microsoft's Windows Workflow Foundation, which in .net 4.0 finally reached a usable status. There are several other companies that provide .NET workflow solutions, such as Skelta and K2. Java, as usual, has many workflow engine options, such as those from IBM, JBoss, and Flux. Oracle offers a workflow package, WF_Engine, along with a Java API, for its database.

Many workflow engines have built-in visual designers to help you model the workflows more easily. Figure 2.12 illustrates a model of the Active Service pattern for report generation build with the Flux Visual Designer tool.

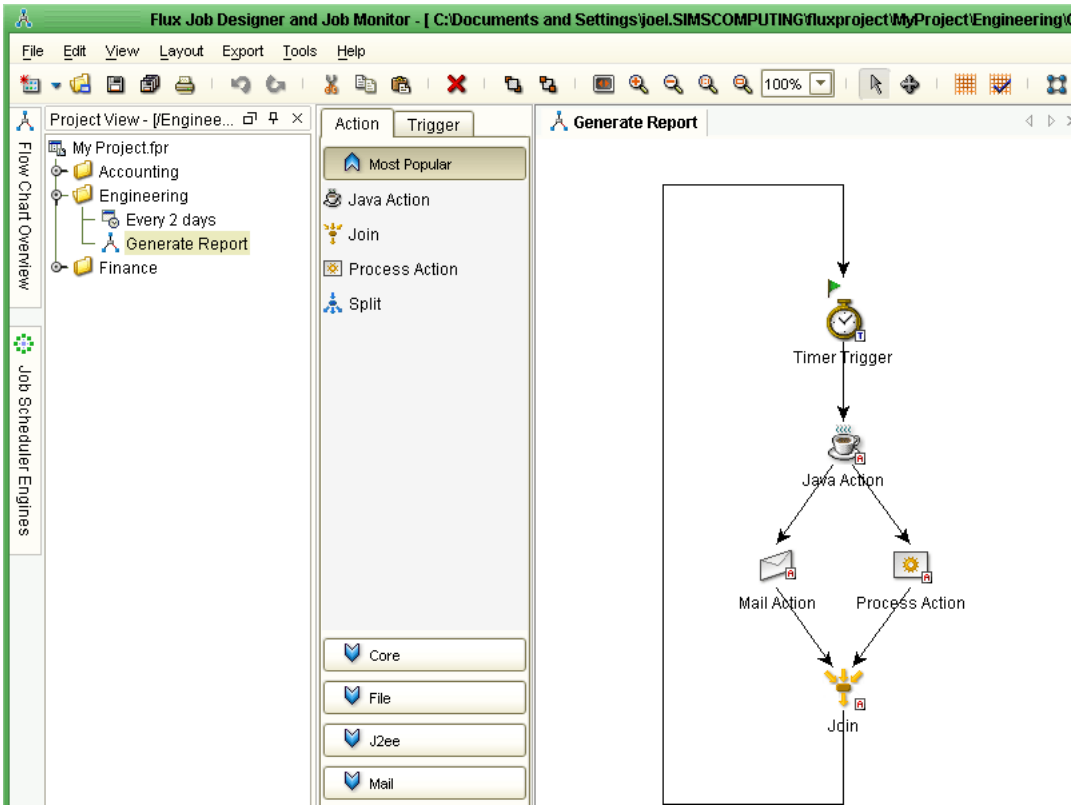


Figure 2.12 – Most workflow engines come with a visual designer tool to help model workflows.

While using a visual designer such as the one in Figure 2.12 is usually the preferred option for modeling flows, you can also specify workflow by hand in XML. Several tools, such as the open source tool, jBPM, support both a designer based and XML based configuration for workflows. Listing 2.1 is an example of a flow modeled in jBPM where we can see a decision point where large orders will be need further approval and smaller ones will go through.

Listing 2.1 : Partial XML of a credit approval workflow implemented for jBPM

```
<start-state name="start">
   <transition to="credit approval"></transition>
</start-state>
.
.
.
<decision name="is user registered?">
```

```
         <handler config-type="bean"
class="org.springmodules.workflow.jbpm31.JbpmHandlerProxy">
               <targetBean>jbpmEvaluateOrderValue</targetBean>
               <factoryKey>jbpmConfiguration</factoryKey>
               </handler>
               <transition name="large_order" to="Review And
Approve"></transition>
               <transition name=”noraml” to="Process Paypal"></transition>
   </decision>
.
.
.
```

Some workflow engines, such as Microsoft BizTalk, or IBM WebSphere MQ Workflow, are better suited to orchestrate interactions between services interaction and not internal workflows, due to their increased complexity (and cost).

2.4.4 Quality Attribute Scenarios

The main benefit of using the Workflodize pattern is added flexibility. Programming a workflow is a visual process (at least with most workflow implementations), which is relatively easy to master. This added flexibility can result in quicker time to market for change requests, leading to greater business agility. Here are two samples of success for the Workflodize service pattern:

Table 2.5 – The quality attributes that can make us consider using the Workflowdize pattern.

Quality Attribute	Practicle Advantages	Sample Scenario
Flexibility	Add new business processes	For a mobile operator, adding support for new a plan goes from weeks to days.
Reusability	Core module set definition	Reuse 90% or more of the of the common sales process activities for each new plan added.

The Workflodize pattern adds a lot of flexibility to a service as you can dynamically change behavior. A different aspect of flexibility can be found in the Edge Component pattern, which we'll take a look at in more detail now.

2.5 Edge Component

The last of the foundation patterns we'll examine in this chapter is the *Edge Component* pattern. The Edge component is classified as foundational because it's a platform used to implement other patterns. Namely, it adds a level of separation on top of business logic that enables a lot of flxibility. Let's examine some, real-world, scenarios from a few projects to illustrate.

2.5.1 The Problem

Scenario 1: Here, we had a requirement to build a common platform for defense solutions. This platform had base services that were reusable in many solutions. For example, one of the core services provides a unified view of military targets. The first implementation built on that platform used a messaging infrastructure based on Tibco Rendezvous®. A second implementation used a different messaging technology altogether (WSE 3.01, in this case). However, both implementations were required to use the same business logic to handle and process the messages.

Scenario 2: In this case--the same one mentioned in the Workflodize pattern in the previous section--a mobile operator needed to introduce new usage plans and offerings on a regular basis. The service interface remained stable, but the business logic kept changing and adapting to the new plans (the opposite of scenario 1).

Scenario 3: Here we had a system that contained many services. Each of these services handled a different business aspect, yet all of them needed to perform common tasks, such as authenticating requests, logging requests in an audit trail, and so on.

Within these three scenarios, we have different concerns, such as business logic, technology choices, cross-cutting features, and so on. Each of these concerns can change independently of the others; therefore we need a way to enable flexibility:

> **How do we allow the service's business aspects, technological concerns, and other cross-cutting concerns evolve at their own pace, independently of one another?**

The easiest option is to *make* new copies of the service features that need to be reused in each scenario; an approach is also known as "own and clone". This obviously creates a maintainability problem as you now have multiple copies of the same business logic or cross-cutting features within several service implementations. Bug fixes and enhancements made to one need to be duplicated across all services, which is a time-consuming and error-prone process. Therefore, this isn't much of a solution at all.

2.5.2 The solution

As discussed earlier, separation of concerns is well-known object-oriented concept used in cases like this. The root principle is known as the Single Responsibility Principle (SRP). SRP states that every class should have a single responsibility, and that all its related methods should be narrowly aligned with that responsibility. Applying this to services, we get the following solution statement:

[1] Microsoft interim solution for the WS-* stack before Windows Communication Foundation

Add Edge Component(s) to the service implementation to add flexibility, and separate the business logic and the other concerns, such as contacts, protocols, technology choices, and additional cross-cutting features.

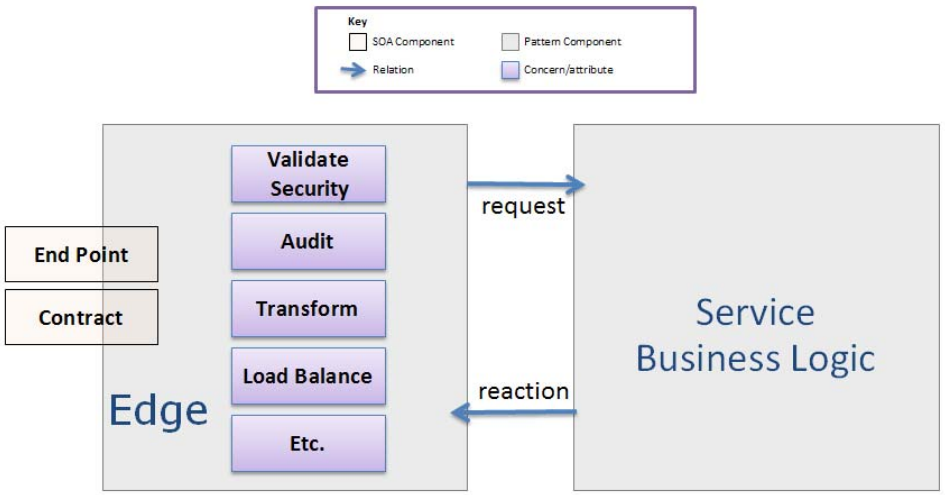


Figure 2.11 – Adding an edge component allows the service itself to focus on the business logic and not extraneous features.

The main idea behind the Edge Component pattern, as demonstrated in Figure 2.11, is separation of concerns. The Edge Component is the place to take care of all the cross-cutting features such as audit, specific endpoints types, contract version mediation etc. that are not part the business logic of the service. The business logic of the service is then handled in a separate component implementation that focuses solely on the business logic and remains free of the other concerns. In a sense the Edge Component pattern provides a façade, or proxy, to a service implementation.

For example you can apply a *pipes and filters* architectural style and chain several classes/component together, each dealing with a specific concern. Figure 2.12 shows a sample use for the Edge Component that applies a validation filter to ensure a message is correctly formatted. Subsequently, a transformation filter translates an external contract format into an internal one. Finally, a routing filter routes the message to the correct component within the service. These sub-components can be reused from service to service as needed, and can change and evolve independently of specific services.

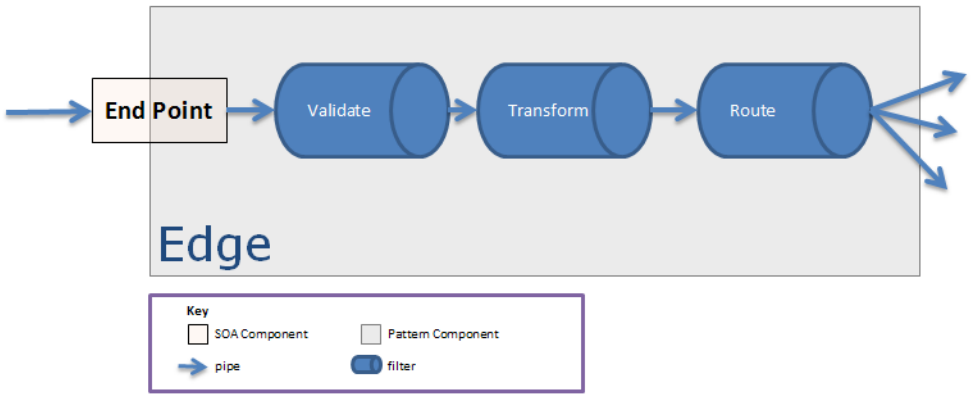


Figure 2.12 – As an incoming message is received, it goes through validation, transformation, and routing edge components used by the service.

The edge component is very useful and I've personally introduced it in most SOA projects I designed. Many of the structural patterns mentioned in this book expand and build on the Edge Component pattern. Let's take a look at the technological aspects of this pattern.

2.5.3 Technology Mapping

Given the wide range of uses for the Edge Component pattern, we have only a few restrictions choosing a technology to implement the pattern and plenty of examples of where we can use it.

For example, both JAX-WS and Windows Communication Foundation (WCF) implement the Edge Component pattern for you, but they only handle lower level concerns called bindings. These concerns are also mentioned in the various WS-* standards. However, you may still need to implement many high-level concerns like routing, contract translations, data transformations, and so on, yourself. An interesting technology option is a Java-based framework called Restlet®.

The Restlet® engine, created by Restlet s.a.s, is a Java library for implementing RESTful services. It has built is classes, such as filter and router, which allow you to easily build edge components. Consider the example in Figure 2.13.

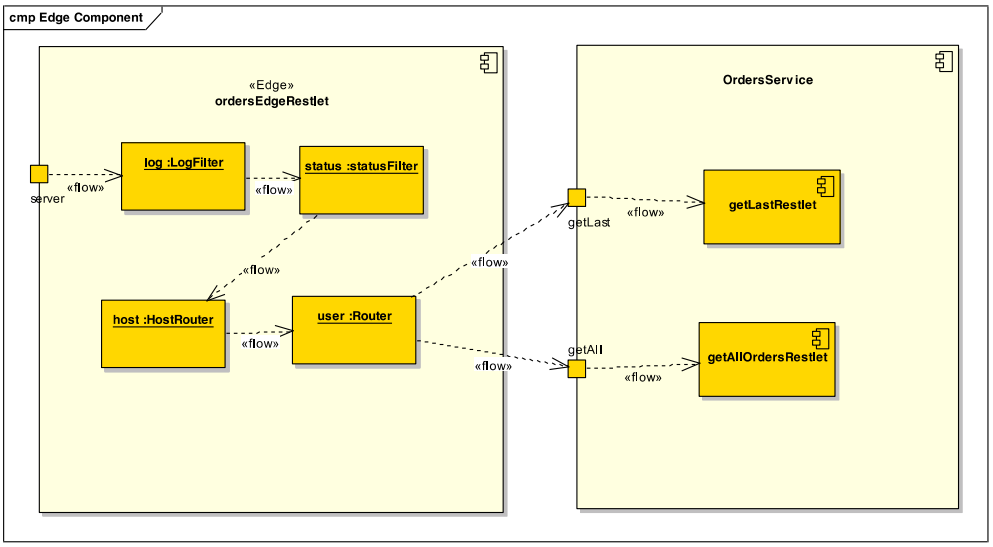


Figure 2.13 – As a request is received ,it gets routed through and handled by the different components before it gets to the actual business service.

Here, we see a possible Edge configuration on an Orders service whose contract has two operations: *getLast* (which returns the last order), and *getAll* (which returns all the orders for a specific customer). However, before the call invokes the business logic, we have to log it, validate data and parameters, enforce security constraints, and finally route the call to the appropriate business component. Adding an Edge component lets us configure and reconfigure this activity without affecting the business logic components.

Another interesting example is the Turmeric SOA framework, which is an open source framework from eBAY.Figure 2.14 below shows the server-side architecture diagram from Turmeric site (see further reading for link). We can see the service implementation as a single (green) rectangle. Most of the diagram explains what is effectively a large Edge component implementation. When a message arrives it passes through a protocol processor and then through an incoming pipeline that handles logging , security and globalization (g11n in the diagram).

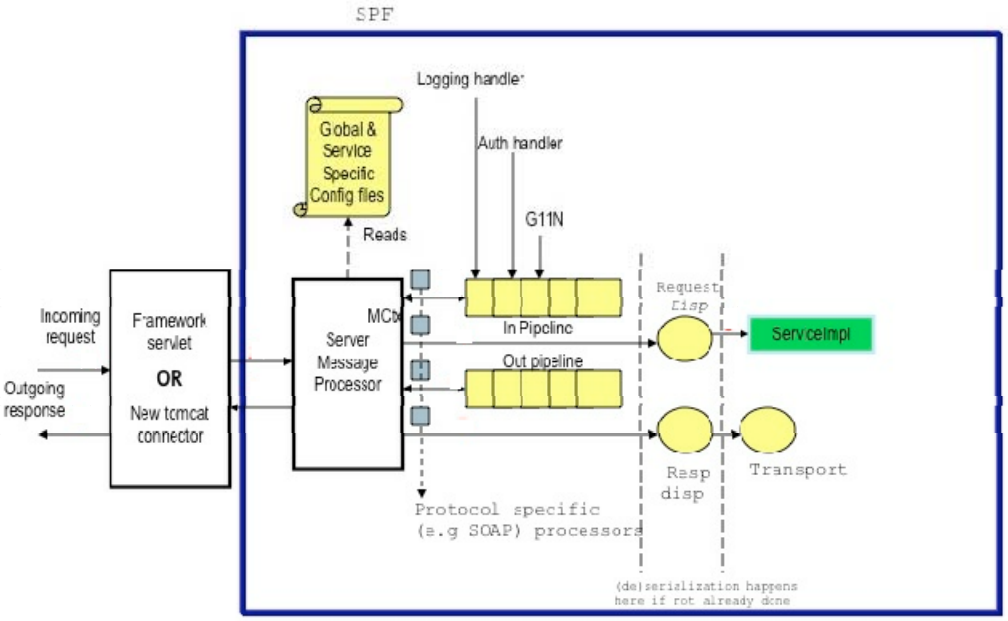


Figure 2.14 Server architecture diagram from Turmeric web site. The actual service implementation is the green triangle on the right. The rest of the diagram shows some of the concerns the edge component implementation handles such as contract endpoint, logging, authorization and globalization (G11N in the diagram above).

As we've seen, the Edge Component pattern is supported by all current technologies, and even implemented internally by some of them. The *Further Reading* section at the end of the chapter contains references to other resources that expand on the technologies mentioned in this section.

2.5.4 Quality Attribute Scenarios

The Edge Component pattern can be associated with two quality attributes: flexibility and maintainability. With it, it's easier to change and enhance the external properties of a service without affecting the business logic. Table 2.6 lists two examples of success for the Edge Component.

Table 2.6 – The architectural scenarios that can make us think about using the Edge Component pattern.

Quality Attribute	Practical Advantages	Sample Scenario
Maintainability	Backwards Compatibility	As contracts evolve, the

		services should be able to support consumers using older versions of the contract
Flexibility	Extension points	Within the next year the customer is expecting to need SOX compliance and add auditing across the board

The Edge Pattern is the last of the foundation structural patterns for SOA. To wrap this chapter up, let's review all of the patterns that we've covered so far.

2.6 Summary

Chapter 2, being the first chapter that presents SOA patterns, deals with few of the foundation structural patterns to build services. These include:

- Service Host – a common wrapper to host service instances and reuse across services
- Active Service – implement at least one independent thread in the service so it can safely call external services
- Transactional Service – handle messages inside a transaction to gracefully recover from error conditions
- Workflodize – add a workflow *inside* the service for added flexibility
- Edge Component – separate interface (contract) from implementation to enable flexibility and maintainability

The next two chapters provide patterns to address additional requirements including scalability, performance, and availability, security and management.

2.7 Further Reading

Table 2.7 resources for further reading on topics covered in this chapter.

Area	Resource name	Why
ServiceHost	Introducing windows server AppFabric – David Chappel	Description of Microsoft appfabric a WCF servicehost
ServiceHost	OSGi in Action - Creating Modular Applications in Java Richard S. Hall, Karl Pauls, Stuart McCulloch, and David Savage	OSGi is a framework fo composable componets that can, and sometimes used as a servicehost container
ServiceHost	Mark Seemann, Dependency Injection in .NET, Manning.	A good book explaining dependency injection which

		is one of the concepts the pattern promotes.
Transactional Service	The Byzantine General problem – Leslie Lamport, Robert Shostak and Marshal Pease	This seminal paper explains the basis of distributed consensus
Workflowdize	Workflow patterns page	A site explaining many of the patterns available for designing workflows.
Edge Component	Restlet engine	*A* REST based framework supportive of the Edge component notion
Edge Component	Turmeric framework	*A SOAP/*REST based framework supportive of the Edge component notion

2.8 Bibliography

Barbara Liskov Data Abstraction and Hierarch [Conference] // Proc. of OOPSLA conference. - 1988.

Liskov substitution principle http://en.wikipedia.org/wiki/Liskov_substitution_principle

3

Patterns for Performance, Scalability, and Availability

We started looking at pattern in Chapter 2 where we looked at some foundation patterns that provide a good starting point for building Services. When we design software architecture for a complete system, we need make sure it will accommodate additional sets of requirements beyond the basics. For instance we need to take care of maintainability, security, reliability, and so forth. One very important quality attribute or requirement class is performance. Performance is actually made of several concerns, such as throughput and latency, which are sometimes are orthogonal and sometimes contradictory.

SOA principles and guidelines don't always help to solve performance problems. In fact, SOA is almost inherently bad for performance by making the components distributed, thus increasing latency and adding layered of indirection. Chapter 3 will present patterns to help mitigate performance, scalability, and availability challenges. Availability and scalability are bundled together with performance as a solution to one quality often helps to resolve the others. For example, one strategy to increase performance is load balancing (see *Service Instance* pattern in section 3.4). If implemented properly, it can also help increase service availability as each load-balanced server provides redundancy for the other(s).

Many people feel that performance, availability, and scalability are easily solved with more hardware. Unfortunately, this is often not the case. This is especially true where new technology or development approaches are involved. It turns out that utilizing additional hardware, implementing load balancing for services, and ensuring adequate application performance when failures do occur, are very difficult problems to solve correctly. Fortunately, when designing SOAs, you don't need to start from scratch. Instead, you can build on the experience and solutions already in place in other environments and

technologies. The challenge, and the topic of this chapter, is to bring this knowledge into the world of SOA while remaining true to the SOA architectural principles and benefits.

Reviewing the architectural components of SOA presented in Chapter 1 (as illustrated in Figure 3.1) you can see that the patterns related to performance, scalability, and availability, mostly have to do with the internal structure of services. Some of these patterns are also related to more than one component of service's interface – namely the End Point and the Contract. It's important to note that, as mentioned in chapter 1, SOA in itself is mainly focused on other quality attributes such as flexibility and interoperability, but doesn't offer much guidance for performance, scalability, and availability.

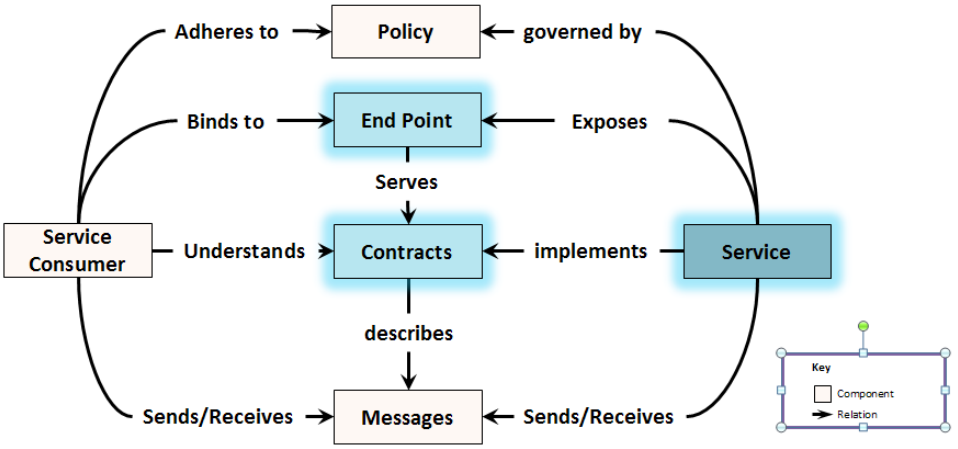


Figure 3.1 – This chapter focuses on availability and scalability patterns for the service itself, as well as the endpoint and contract components of SOA.

Table 3.1 lists the patterns discussed in this chapter and the problems that they address.

Table 3.1 – list of patterns discussed in chapter 3

Pattern name	Problem addressed
Decoupled Invocation	How can a service handle normal request loads, peak request loads, and a continuous period of time at high-load without failing?
Parallel Pipelines	How can we build services that maintains state and high throughput?
Gridable Service	How can we provide services with location transparency, and gracefully recovery from failure, that would not affect consumers?
Service Instance	How can we provide services with location transparency, and gracefully recovery

	from failure, that would not affect consumers?
Virtual Endpoint	How can we provide services with location transparency, and gracefully recovery from failure, that would not affect consumers?
Service Watchdog	How can we increase availability, identify and resolve problems and failures that are service specific,?.

The first pattern we're going to look at is the *Decoupled Invocation* pattern, which serves as a base pattern on which other performance related patterns can build. Let's take a look at this pattern in more detail now.

3.1 Decoupled Invocation

I've mentioned in chapter 1 that SOA helps reduce coupling between components (services) by putting a lot of emphasis on the interface. As we've seen in the discussion on the Active Service pattern in chapter 2, this doesn't take care of temporal coupling, or the problems related to time. Another aspect of temporal coupling is the request/reply pattern (see chapter 5), which is what most SOA implementations use.. With request/reply, we typically expect the service to return a result immediately. This couples the consumer to the service in time, and can produce an excess amount of load on services. This can result in a performance bottleneck, as the maximum load is the maximum number of requests the service can handle concurrently. Let's illustrate this with an example.

3.1.1 The Problem

Consider an on-line music store. Let's say that the backend system has (amongst others) two services: one that deals with album orders and one that deals with single-track orders – see Figure 3.2. The left–hand portion of the diagram illustrates a normal business day for this store with a mild load on both services; purchase requests are well distributed in time. The right–hand portion of the diagram shows what can happen on a day that some crazy hit is released. The same store suddenly has to handle a much higher number of purchase requests than normal.

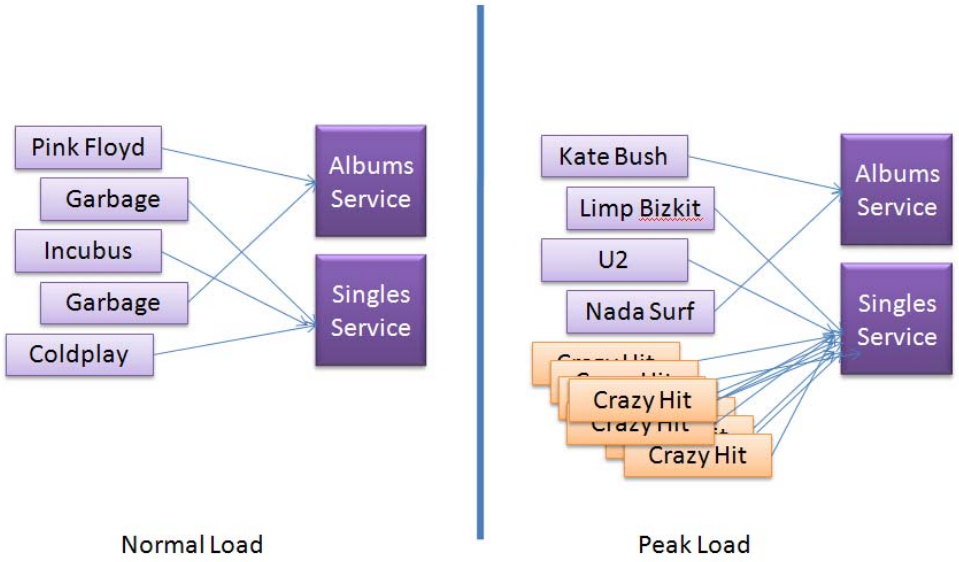


Figure 3.2 – This diagram demonstrates a music store's service-request loads under normal conditions versus peak loads when a popular song is released.

Obviously, the music store needs to be able to handle all incoming requests, even under high load, or customers will take their business elsewhere. On the other hand, it's important that the service be built economically to handle normal loads, but still be able to handle peak loads without failing.

How can a service handle normal request loads, peak request loads, and a continuous period of time at high-load without failing?

One option is to estimate the peak loads and deploy enough server power to ensure you can handle them. One problem with this approach is that it wastes money and resources; servers may remain idle during normal operation, used only during rare bursts of activity. The idle computers have purchase, maintenance, and operational costs. A bigger problem is that, often, much of the service's processing may be out of your control. For instance, external credit card clearing service requests, shipping requests, and so on, may fail under load, or slow down your internal service response times. Finally, you may find the need to prioritize some requests over others. You can set the overall quality of service (QOS) parameters according to the most demanding request type – but then you may need more resources to be able to handle your steady on going load.

A good solution for this problem is deploying to a cloud provider like Amazon, Microsoft's Azure or VMWare's Cloud Foundary and elastically grow the number of servers in peak load. One problem with this is that you need to make sure your service is cloud ready (something you should probably take care of anyway). The more serious problem is that it will take care of the scaling to peak loads parts but it wouldn't completely cover the without failing part.

What we need is something that will enable us to register requests quickly and reliably and free server resources to handle new requests. The solution should provide this, while letting the requestors know that their request is going to be handled. This is what the Decoupled Invocation pattern is all about.

3.1.2 The Solution

When a new request enters the system, instead of immediately invoking the business logic we can do the following:

> **Utilize the decoupled invocation pattern and separate replies from requests: Acknowledge receipt at the service edge, put the request on a reliable queue, then load-balance and prioritize the handler components that read from the queue.**

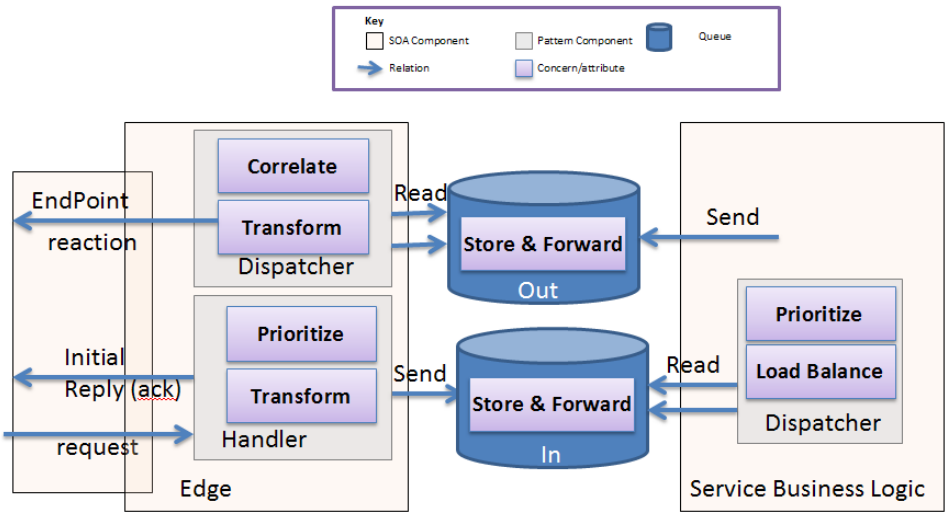


Figure 3.3 – The Edge component contains a Handler that accepts incoming messages, acknowledges them, and then queues them. The business logic then reads the queued messages at its own pace. The queue is also used for the responses.

As illustrated in figure 3.3 above, the Decoupled Invocation pattern is composed of three basic components: a *Handler*, a *Queue*, and a *Dispatcher* that mediates between them. Here's how the initial request processing works:

- The Handler listens for incoming requests from the End Point.
- When a new request arrives, the Handler sends an acknowledgement to the sender.
- The Handler is responsible to the initial treatment, or preprocessing, of incoming messages. This may include message transformation, or prioritization, based on knowledge it infers from the messages themselves. Overall, this processing should be kept minimal, as the goal is to be able to quickly queue and acknowledge incoming requests.
- Lastly, the message is put onto a Queue.

The Queue, which is the second component of the Decoupled Invocation pattern, stores incoming messages, and allows the service to consume the messages at its own steady state thus overcome peek loads.

Furthermore, we can set up the queue to be persistent and the service will not lose any requests it already acknowledged even if a catastrophic server failure will occur. If the queue is transactional, you can implement the Transactional Service pattern (see chapter 2) and increase the overall robustness of the service even further.

The Dispatcher is responsible for creating as many reader components as are needed for the current request load, which is measured as the number of messages waiting in the queue. The Dispatcher can also prioritize incoming tasks based on internal consideration, such as resource availability. The dispatcher is a good point to introduce elasticity if the latency of handling the messages is also important. (see the LMAX architecture in the further reading section)

Recall that the Handler can acknowledge the request as part of the preprocessing. However, it's usually best to do this inside an Edge Component (see Edge Component pattern in chapter 2). This helps ensure that the service-processing load is kept to a minimum, allowing it to process requests as efficiently as possible,

WHEN TO ACKNOWLEDGE IN THE SERVICE

Often, request-acknowledgement processing requires some extended business logic to be executed. In this case, you need to consider whether the response is tied to the contract, or if it's tied to the service's core business. If it's related to the core business, acknowledge the request from the service implementation, otherwise acknowledge it from the Edge component.

Placing requests on the Queue is a relatively low-cost operation that can be performed efficiently, making this portion less susceptible to failure during peaks. The actual handling of the incoming requests can be performed at a reasonable pace, dictated by service resource

availability and overall load. Load balancing can be achieved by running multiple readers working against the Queue. Note that this works well for peak loads, but if you have continuous high-request loads over an extended period of time, you may need to alter your approach or the request Queue may overflow. See the *Parallel Pipelines* and *Gridable Service* patterns later in this chapter for strategies to help with continuous loads. Further, configuring the Queue to be priority-based (or configuring several queues according to priority) allows you to maintain different levels of QOS for different message types/different contracts.

The Decouple Invocation pattern is a good way to implement the *Request/Reaction* service integration pattern, discussed in Chapter 5. Additionally, since the reply is delivered to the consumer as a new message, it's recommended that you correlate messages by adding an identifier that is returned to the consumer on the acknowledge message as well as the final reaction Using correlation id, you can help the service consumer understand the reaction is related to a request it sent earlier.

Let's take a look at few of the options for implementing the Decoupled Invocation pattern using the currently available technologies.

3.1.3 Technology Mapping

To implement this service, the underlying messaging technology needs to support store-and-forward queues, preferably with persistence and transactional support. Most enterprise messaging middleware packages support this, such as Microsoft Message Queue™, WebSphere MQ™, Sonic MQ™, and any other JMS-compliant implementation. Another class of messaging solutions are AMQP based queues like RabbitMQ and Apache Qpid (where the advantage of AMQP is that it is also a wire standard which means you can integrate different implementations easily)

One point to consider is whether you really need messages to be persistent; if you don't, you can just use in-memory queue as long as the service and the edge run in the same process. If they're in separate processes, most message message-oriented middleware supports express message delivery (without persistence) for faster performance. Furthermore, you need to consider transaction support, where you'd require a queue that supports distributed transactions. If so, consider combining Decoupled Invocation with the Transactional Service Pattern described in Chapter 2.

Another issue to consider is that the reply will be sent asynchronously. You need to establish a bi-directional channel in order to do that. Messaging is a good option, and consistent with our approach so far. However, you can also use AJAX technology, which lets you *push* content to the client. In cases where acknowledgement and/or reply messages aren't required, you can define the contract to support One-way messages. Consider, for example, the simple code excerpt below, using Windows Communication Foundation (WCF):

```
[ServiceContract]
interface PurchaseSongs
{
   [OperationContract(IsOneWay = true)]
```

```
    void SubmitOrder()
}
```

The attribute on the SubmitOrder operation tells WCF to just send the message without returning a reply. You can use one-way messages if you don't care too much for the reliability of the message (e.g. it is a cyclic message, where if one is lost the next one will compensate), or if you're using a reliable transport. As usual, choosing the right technology boils down to which of the quality attributes are most important to you. In this case it's a potential performance versus reliability trade-off.

3.1.4 Quality Attribute Scenarios

The Decoupled Invocation pattern helps solve the potential performance bottleneck presented in the problem section. It does this with a queue between the caller and the message handler component. Placing a message on the queue is an efficient operation, which means the service will be free to accept new requests sooner. If you keep the Handler simple, you can employ the Virtual Endpoint patterns (see section 3.5 later in this chapter) to also resolve availability problems when faults occur.

Additionally, since requests are handled asynchronously, the Decoupled Invocation pattern can help increase service flexibility, as coupling between the service and its consumers is reduced. Just as importantly, the Decoupled Invocation pattern helps with testability. Table 3.2 shows a few examples for scenarios and quality attributes that the Decoupled Invocation pattern can help with.

Table 3.2 – This table lists the architectural advantages of using the Decoupled Invocation pattern.

Quality Attribute	Practical Advantage	Sample Scenario
Performance	Eliminate data loss	No message acknowledged by the system will be lost
Performance	Decrease latency	Handle incoming requests without degrading latency, even under peak loads
Testability	Increase isolation	A service can be tested in isolation from the services it interacts with
Flexibility	Reduced assumptions	For normal interactions, invoke services in a fire-and-forget manner

While the Decoupled Invocation pattern enables growth, scalability, and performance, the *Parallel Pipelines* pattern (discussed next) builds on it to increase overall service throughput.

3.2 Parallel Pipelines

The Decoupled Invocation pattern helps to handle peak loads by queuing up requests and deferring processing to low-peak hours. However, this solution doesn't increase overall service scalability when increased request rates are maintained. For instance, under a continuous high request rate, the request can accumulate in the queue and eventually overflow. We need another strategy to handle continuous loads. Let's start with an example.

3.2.1 The Problem

Consider a credit card clearinghouse, sometime known as transaction processing service. Figure 3.4 illustrates a basic processing flow that takes place when a credit card purchase request arrives.

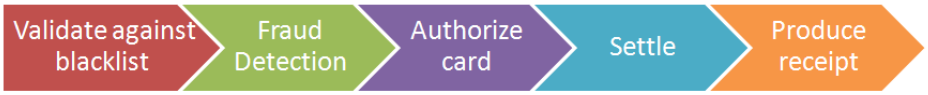


Figure 3.4 – First, check that the card and IP address are not flagged. Next, try to infer from the card's past activity if this purchase may be fraudulent. Finally, authorize the card against the bank, perform the transaction, and produce a receipt.

As illustrated, the processing for a credit card transaction begins with a check against known blacklists (bad card numbers, bad source IP address, and so on). Next, the service looks for fraudulent patterns in the transaction. Finally, if everything checks out to this point, authorize the card against the card issuer, settle the account (make the actual payment), and produce a receipt. Naturally, if one of the checks fails, the processing enters an exception-processing path (not shown).

The primary problem concerns the number of steps in the process. As a secondary problem, some of the steps involve communication with external services. We may have difficulty getting a service such as this one to scale. Here's the problem statement:

How can we build services that maintains state and high throughput?

One solution is to introduce concurrency (multiple threads) and have each request run in its own thread, or from a thread pool. However, multi-threaded programming is complex, more difficult debug, and introduces performance and scaling issues of its own.

A couple of possible solutions using other patterns:

- Introduce concurrency, and use the Service Instance pattern (discussed later in this chapter), and deploy to multiple load-balanced servers. Since the service is stateful, however, this won't work unless the state is synchronized and replicated across all servers.

- Use the Gridable Service pattern (discussed later in this chapter) and introduce a computational grid. However, this solution itself is very complex, and doesn't work well when external service calls are involved.

Another possibility is to use the Parallel Pipelines pattern

3.2.2 The Solution

To maintain high-throughput, and be able to work with stateful components, it's suggested to use the following strategy:

Implement the Parallel Pipelines pattern, where you break the process into sub-tasks, add a queue between them, and make each sub-task an independent component.

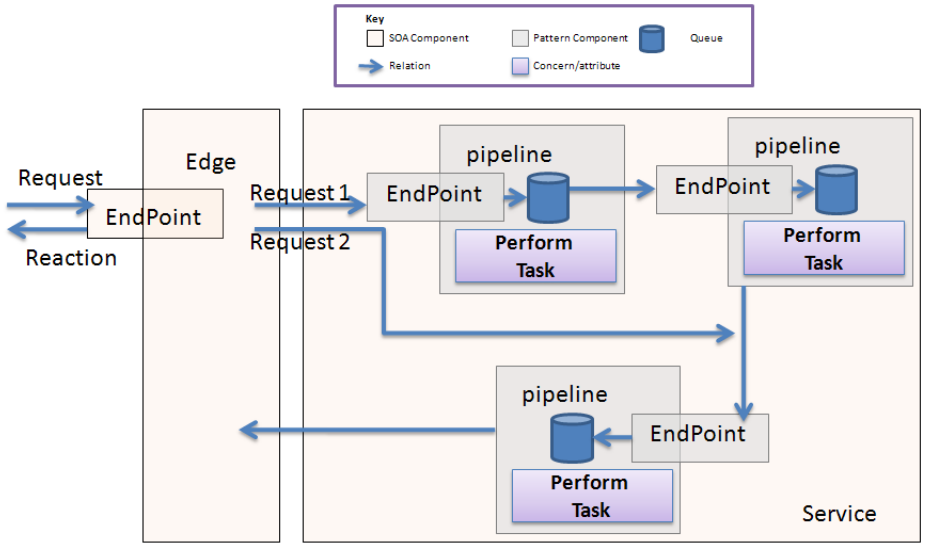


Figure 3.5 – With the Parallel Pipelines pattern, the processing is broken into sub-tasks that are connected by queues to form a processing pipeline. Note that different requests can have different flows of tasks.

The Parallel Pipelines pattern, as Figure 3.5 demonstrates, is an application of the Pipes & Filters architectural style (see further reading) in the context of SOA. The pipes represent the message transport, and the "filters" are the components that handle the sub-tasks. A pipeline begins with an endpoint where the messages arrive. The incoming messages are placed in a queue, and the pipeline services the queue as efficiently as possible. Each component in the pipeline works with the message, and sends the results onto the next component via its outbound queue. Some components can maintain more than one outbound

queue, depending upon the result of message processing. With this paradigm, you can orchestrate alternative pipeline processing paths based on request message content, and variations in processing along the way.

The following are advantages of the Parallel Pipelines approach:

- The pipelines pattern is relatively simple to implement.
- Pipelines are easy and test as they operate independently (you use the same technologies and principles you did for the services than include them).
- Since the overall problem is broken into sub-tasks, each pipeline component tends to be simpler.
- To scale the solution, you can distribute the pipeline across as many servers as needed.
- When you need to scale the solution the simplest option is to put each pipeline on its own server.

When deciding how to divide the process into pipelines you can either make sure that the pipelines are independent from each other or make sure you pass the needed context between the pipelines, e.g. as a document which gets more and more context as it passes the steps.

The Parallel Pipeline pattern also works well in combination with the other performance and scalability patterns we'll discuss in this chapter. For example, you can use Parallel Pipelines with the Gridable Pattern (see section 3.3) to solve a performance problem within one of the sub-task components.

The challenge is to partition the process in a way that's easy to implement and deploy, and still fulfills the business goals of the parent service. It is preferable to partition according to business boundaries, so that each pipeline is a business service in its own right. However, it's also acceptable to partition the pipelines according to a technical need, just try not to expose this partitioning to external callers. For example, Figure 3.6 shows the credit card clearing house modeled with the Parallel Pipelines pattern.

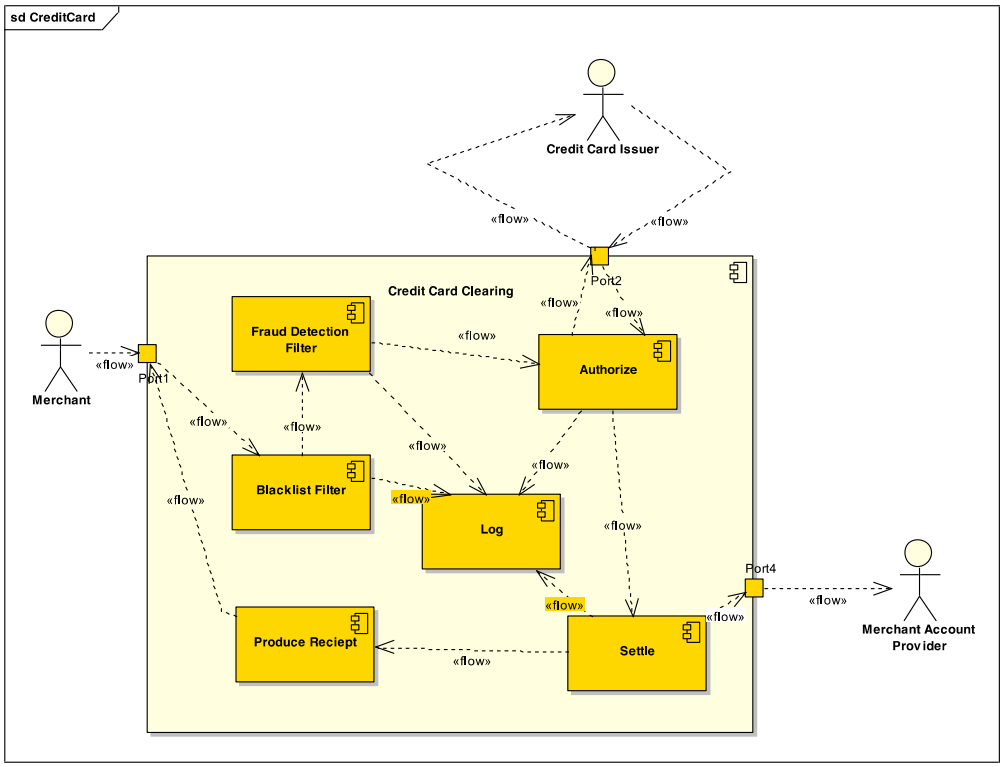


Figure 3.6 – Looking at the credit card clearing house service from the previous example, each task is modeled as an independent service. The Parallel Pipeline pattern has improved the overall scalability of the design.

As the figure illustrates, each sub-task (blacklist, fraud detection, and so on) is modeled as an independent component in an overall pipeline. Each is responsible for just one task, which it can perform relatively quickly. Since we have six pipeline components, we can handle around six different messages in different stages of the pipeline simultaneously without the need to introduce concurrent programming within each component. Contrast this to a monolithic service, where each new request needs to wait for the previous request to be processed in its entirety before beginning.

If we look at the different pipelines that make the process in figure 3.6 above we can see that most of the pipelines are self-contained so they can handle the input without dependencies on other pipelines or external resources. An exception to that is the Authorize pipeline which needs to communicate with an external resource to complete its work. We can see here another advantage of this patterns as now instead of making a lot of small requests

to the external resource (each with the overhead of serialization, network, security etc.) we can make a chunkier call for batch of requests which is more efficient, while still handling additional request with other parts of the process (by other pipelines)

You can orchestrate the different pipeline components with the Workflodize Pattern (discussed in Chapter 2), along with a workflow engine, such as Jboss jBPM, to drive the message flow through the pipeline. The easiest way to track and understand the state within the pipeline is with the Transactional Service pattern (also discussed in Chapter 2). This ensures that each pipeline component performs a discrete unit of work in isolation.

3.2.3 Technology Mapping

As mentioned above, you can partition pipeline components, or set their boundaries, based on technical considerations, in addition to business needs. This sort of partitioning is acceptable as long as the overall service is exposed via an Edge that implements a meaningful contract, and the sub-component breakdown is not exposed to the caller. Also, be careful not to partition the service into too many components, or with components that are too fine-grained, as they may become difficult to manage and make the latency unbearable.

Implementing the parallel pipeline pattern is not too complicated (the design of which operation to group in which subcomponents it the complicated part) .For instance you can use Akka actors – a Scala framework also usable from Java, that lets you implement remote message passing between components. Another option is to base a solution on JavaSpaces technology , which has commercial implementations like Gigaspaces (usable both form Java and .Net). The nice feature of both of the above mentioned technologies is that, though different, the both allow making components local or remote by configuration and thus partition the logic into pipelines according to your needs/performance requirements.

As usual, we'll finish the discussion of the pattern by looking at some of the motivations to use it.

3.2.4 Quality Attribute Scenarios

Remember that performance is a multi-dimensional trait, and one that is relative by nature. Therefore it's sometimes hard to define clear acceptance criteria. Also, some of the sub-categories of performance can contradict one another. For example, to decrease the latency of message processing, you can choose to forgo transactions. However, this increases the chances of data-loss; clearly there's a tradeoff here. With the Parallel Pipelines pattern, there's a trade-off between throughput and latency. With every pipeline you add, you increase the parallelism in your application and throughput increases as a result. However, this approach can also increase overall message processing latency.

However, the benefits typically outweigh the tradeoffs. First, this pattern helps tremendously to increase service scalability. Additionally, pipelines increase testability; since the service's tasks are independent components, you can test them independently as well. Table 3.3 outlines some of the quality attributes and benefits of the Parallel Pipelines pattern.

Table 3.3 - These are the quality attributes that result from the implementation of the Parallel Pipelines pattern.

Quality Attribute	Practical Advantage	Sample Scenario
Performance	Higher message throughput	Dramatically increase requests per second
Scalability	Handle increased Loads	To handle increased loads, solve the problem with additional servers with no software changes
Testability	Increased component isolation	Testing small, individual, components helps to assure their success when connected in a pipeline

For the sub-tasks within a pipeline that are computationally-intensive, you may need to apply other strategies to keep the service scalable. One such strategy is the *Gridable Service* pattern, which we'll explore now.

3.3 Gridable Service

One characteristic of SOA is that it's built for highly distributed systems. Each and every service is a sub-system in itself; it can run on its own machine and can be located anywhere in the world. Often, services *need* to be distributed to help with computationally intensive tasks. Let's try to understand this with an example.

3.3.1 The Problem

A few years back I managed the biometric product line of a defense systems company. One of the products we developed was a multi-modal biometric platform. Such a system is used to authorize visitors as they enter a secured building, or a secured area. This is a straightforward scenario, as you usually deal with a finite number of people, and each person is equipped with an appropriate identification badge. In this case, the system looks up the visitor's credentials in a database, run some sort of biometric algorithm, and verifies the person's identity. However, the same platform needs to work in other, more complex, scenarios. For example, consider a forensics system where you have a fingerprint collected at a crime scene, where you don't necessarily know who the person is ahead of time. As a result, data is searched against a much large database, which can contain millions of records. If you have more than one modality say finger prints and DNA, then the problem quickly multiplies. In the end, you need to aggregate the results sets from all the searches. As a result, the processing throughout the system can become quite intense.

Multi-Modal Biometrics

Biometrics is one solution to identification and security. It combines something you know (a password), something you possess (an identification badge card), and something that is part of you (biometrics). This includes a fingerprint, face recognition, or iris recognition for example. Multi-modal biometrics involves the combination of results of two or more biometric modalities. The added complexity comes from the algorithm to aggregate the results of the different biometric engines.

Other examples of computationally intense tasks are financial calculations, and simulations systems. Whatever the process entails, the same problem statement applies:

How can we build services to handle computationally intense tasks in a scalable manner?

One option is to scale up i.e. get a larger, stronger server to solve the problem. However, using a stronger server at a problem will work to an extent. Throwing hardware at the problem can also get costly fast. With the need to build redundant systems for failover and load balancing, the cost only multiplies – for most organizations this is not a feasible option and the more cost effective way is to scale out instead.

3.3.2 The Solution

Scaling out, when it comes to computationally intensive tasks usually means to:

Introduce grid technology to the service, with the Gridable Service pattern, to handle computationally intense tasks.

Figure 3.7 illustrates the solution. The Gridable Service pattern is based on a computation grid, and possibly a data grid, as part of the internal structure of a service. When the service business logic needs to handle a task that is computationally intense, the business logic creates a job on the grid root. A Job is made of one or more tasks that can be queued and executed on the grid. The scheduler distributes the tasks to one or more nodes, depending on the job type, where the grid agent executes them.

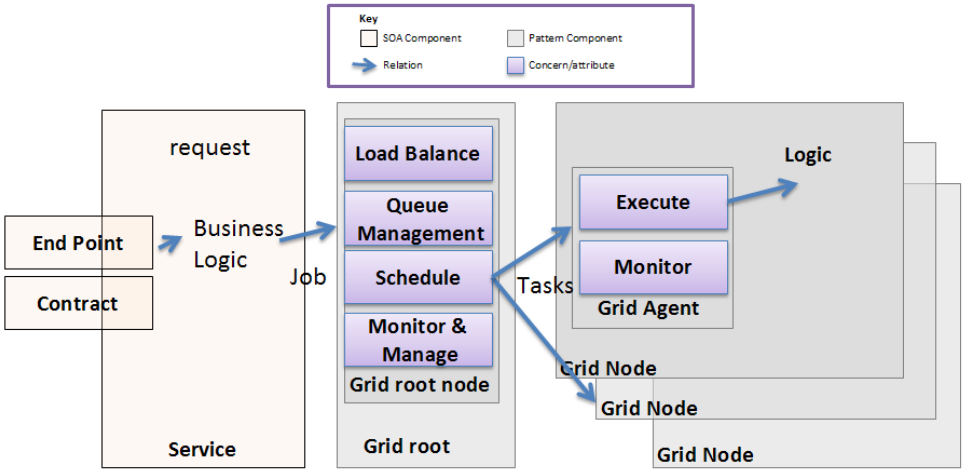


Figure 3.7 – When the business logic within the service has to invoke a computationally intensive task, it creates a job on the Grid root. The Grid root manages all the resources within the grid or the compute cluster and executes the task efficiently.

The Grid infrastructure components (the Agent, root node, and so on) constantly monitor resource availability. Adding additional hardware, configured with the grid components, enlarges the pool of available resources. The grid takes care to maximize the usage and does that based on the load of the machines. This “smart” resource allocation helps solve both scalability and load balancing requirements. Additionally, the grid implements redundancy and failover to provide additional improved if a node fails. The Gridable Service pattern can be combined with the Workflodize pattern (see chapter 2) by making the tasks in the job workflow instances or by having a workflow drive the jobs.

Let’s review the biometric problem presented earlier. One of the services defined in this system is a Pattern Matching service, which takes a biometric pattern (sort of a hash for a biometric sample) and searches for matches in the patterns database. This is a potentially time-consuming effort as the database may contain large numbers of records. Also, you need to use a biometric engine to compare the templates, as some information is more important than others. For example, the distance between the eyes is more important than a beard for a face recognition scenario. Figure 3.8 shows how the problem can be solved using the Gridable Service pattern.

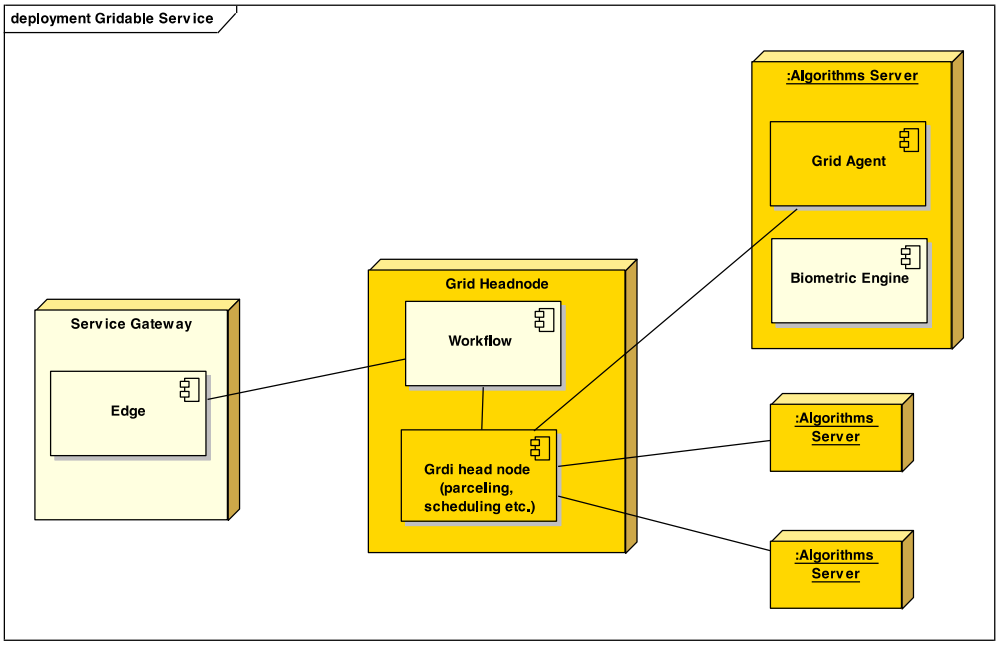


Figure 3.8 – With the Gridable Service pattern applied to one of the services of a biometric platform, the different biometric engines are deployed on the grid, and a workflow drives their invocation.

The edge component translates the request to an internal representation and invokes the workflow that deals with matching. Next, the workflow component works the with grid head node to partition the matching job and schedule it. The grid infrastructure takes care of finding free Algorithm servers, and then invokes the appropriate biometric matching engines.

The Gridable Service pattern helps us solve our computationally intense tasks, but it sounds like a lot of work to implement this pattern. Fortunately, there are quite a few grid implementations available; all we need to do is to integrate them into our SOA. Let's take a quick look at some of the available technology options

3.3.3 Technology Mapping

There are many Grid implementations – all of them can be applied in an SOA context to implement the Gridable Service pattern. One notable effort in grid computing is the Gridbus project, which defines open-source specifications, an architecture, and a reference Grid toolkit implementation for a Service Oriented Grid.

In Grid scenarios, you simply create remote threads of execution; without needing to know where the execution will take place. The grid infrastructure optimizes task execution across connected nodes, based on the available resources across them, and executes each

job on the appropriate machine. Figure 3.9 is a screenshot of the system console for Alchemi.Net, which is a Microsoft .Net implementation of the Gridbus standards.

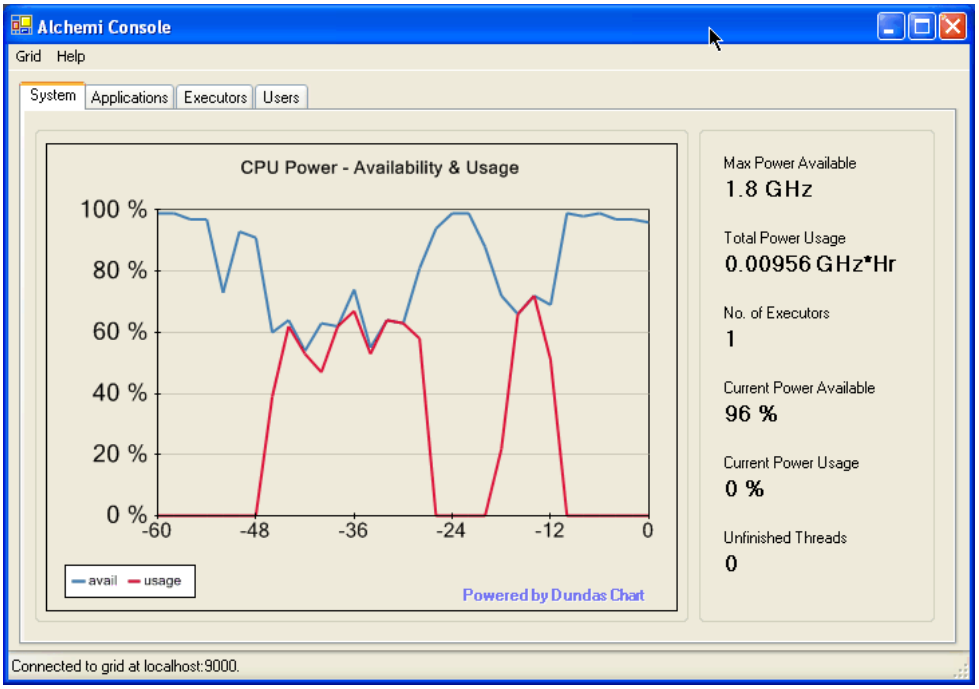


Figure 3.9 - The Management Console of Alchemi.NET a Microsoft .Net based open source project implementing GridBus.

The grid software manages all member resources, and can provide metrics on how the system is doing overall. The same information is used by the Grid internally to distribute jobs efficiently. Gridbus is, of course, not the grid implementation available, such as:

Microsoft Windows HPC Server 2008, which scales your application logic out to thousands of processing cores across your existing Windows infrastructure. See http://www.microsoft.com/hpc/en/us/default.aspx for more information.

A similar notion to pure grid solutions are data grid technologies, there are several options for that in the Java world with products such as GridGain, Hazelcast and others. Data grid solutions, as opposed to computational grid solutions, also hold portions of the data and usually support some sort of map/reduce algorithms where the computation is sent to the data is only moved to perform summaries (reduce).

The WS-* stack of web service protocols also addresses grid design, and there are a few protocols bundled under the name WS-ResourceFramework (WSRF). Table 3.4 shows the five protocols that WSRF is composed of.

Table 3.4 – The five protocols included in the WS-ResouceFramework specification.

Protocol	Description
WS-Resource	Defines the relationship of a resource on the grid to web-services
WS-ResourceProperties	Defines a protocol to retrieve and set the list of features/properties of each resource
WS-ResourceLifetime	Defines the semantics to control the life time of a resource
WS-ServiceGroup	A standard for defining a collection of resources
WS-BaseFaults	A standard for handling problems and faults

To wrap this subject, let's remind ourselves the motivations for utilizing the pattern.

3.3.4 Quality Attribute Scenarios

The Gridable Service pattern, and the grid technology it's built upon, can help with some of the common quality attributes most projects face, such as performance and availability. All of the quality attributes are met with using the same mechanisms that allow redistribution of computational loads based on the available resources. Scalability is addressed by the fact that resources are pooled and constantly monitored. The gird is able to reroute work in case of failure, as well as redistribute the load when a new node is added. Table 3.5 includes a few sample scenarios and benefits of using the Gridable Service pattern.

Table 3.5 – These are the quality attributes and scenarios of the Gridable Service pattern.

Quality Attribute	Practical Benefit	Sample Scenario
Performance	Improved latency	Under normal conditions, completing service requests requires less time
Availability	Hardware failure resiliency	Upon a server crash, the system will remain operational
Scalability	Scale-out	It would be possible to deal with increased service loads with more hardware
Budget	Contain hardware costs	The grid allows you to spread load over less-expensive hardware

One important quality attribute that is missing here is security, as it's not a core capability of the grid. However, serious grid implementations should address security to some degree. The Gridable Service pattern can help you solve some of the basic needs of distributed systems, such as performance and availability. The grid can also help achieve scalability, but Grids are not the only solution here. Let's take a look at another pattern that will also help with scalability.

3.4 Service Instance

So far, we've discussed two patterns that can be used to achieve scalability: Gridable Service, and Parallel Pipelines. To see why we would need another one, let's examine a sample scenario

3.4.1 The Problem

You might remember the blacklist service from the credit card clearing house example mentioned in the Parallel Pipelines pattern (see section 3.2 earlier this chapter). The blacklist service is responsible to verify that the various attributes of an incoming request are not in a pre-known list (a "blacklist") of invalid items. Let's look at the VerifyRequest operation provided by the service – see Figure 3.10 below. Under even normal conditions, the service should experience a high number of incoming requests per second. Each will need to be validated very quickly.

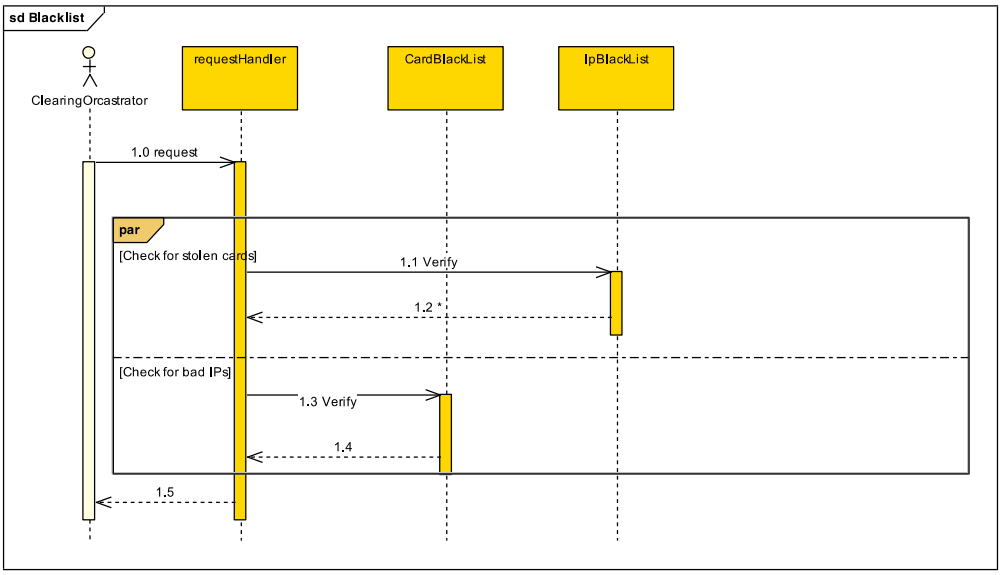


Figure 3.10 – This sequence diagram for the VerifyRequest operation outlines the steps to verify credit card purchase requests.

The Blacklist service is straightforward; it communicates with a database and cache, and verifies that the requester is not in any known lists. One very important aspect of this service is its ability to scale. The problem statement is:

How can we build services that are scalable in a simple and cost-effective way?

Two possible solutions are to use the Gridable Service pattern, or the Parallel Pipelines pattern we covered above, possibly even together. The Gridable Service pattern, though primarily targeted at computationally intense tasks, can essentially solve most of the scalability needs. However, using grid technology can be relatively complicated and expensive. You might want a more lightweight alternative to scalability. The same is true for the Parallel Pipelines pattern. We can isolate each blacklist in its own pipeline, but this can create additional overhead for a relatively simple operation. It may even create an unacceptably large amount of latency for each request.

3.4.2 The Solution

Let's look to apply a more simple solution to a potentially complex problem:

Implement the Service Instance pattern by deploying multiple instances of the service business logic.

As illustrated in Figure 3.11, the Service Instance pattern is built on a simple concept: you deploy multiple copies of the service. Using a dispatcher on the Edge, you distribute the work to the different instances. Depending on the technology you use – you might not even have to implement anything in the dispatcher.

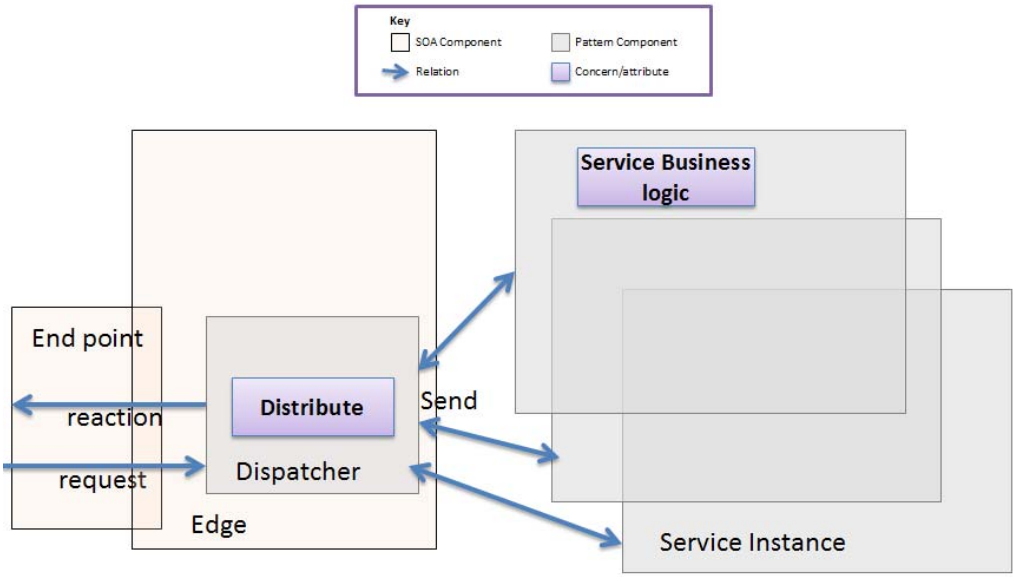


Figure 3.11 The Service Instance Pattern. A dispatcher (usually deployed on the edge) routes messages to one of the instances of the service business.

It's better to maintain a single endpoint and then divide the request load between the service instances. You can build on the Virtual Endpoint pattern if you need multiple endpoints. The important point is that consumers of the service will be unaware and unaffected by the scaling that occurs inside the service (see sidebar for more information)

SCALING INSIDE VERSUS OUTSIDE OF THE SERVICE

When scaling is implemented outside a service, the service is not aware of the scaling is taking place. Multiple instances of the service are deployed on the network. When scaling is implemented inside the service, which components outside of the service aren't expose to how the scaling occurs.

In most cases, it's recommended to scale inside the service, as it hides the complexity from the consumers, which makes for easier maintenance and integration. It also lets you treat the service as an independent system, and increases the overall autonomy of the service. Lastly, scaling outside the service requires that the service business logic will be stateless, which is not always possible.

The Service Instance pattern is best suited for stateless service implementations. If you have state that needs to be shared between the instances, then you should probably look to use the Gridable Service pattern.

The *Decoupled Invocation* pattern is related to the Service Instance pattern. To combine the two, you implement the service instances as multiple readers that process the same input queue (see Decoupled Invocation pattern earlier in this chapter for more details).

3.4.3 Technology Mapping

Implementing the Service Instance pattern does not require a particular technology. Instead, you implement a Dispatcher in the language of your choice, and distribute requests to the farm of servers running your service. This is especially true if you implement this pattern on top of the Decoupled Invocation Pattern.

An alternate, and probably a more common way, to implement the Service Instance pattern is to build on the Virtual Endpoint pattern (described in the next section) and use one of the available load balancing technologies. You can implement this at the application level with packages such as Apache JServe, or at the OS level with packages such as Microsoft's NLB Cluster (see figure 3.12), or one of the linux options like HA-Proxy.

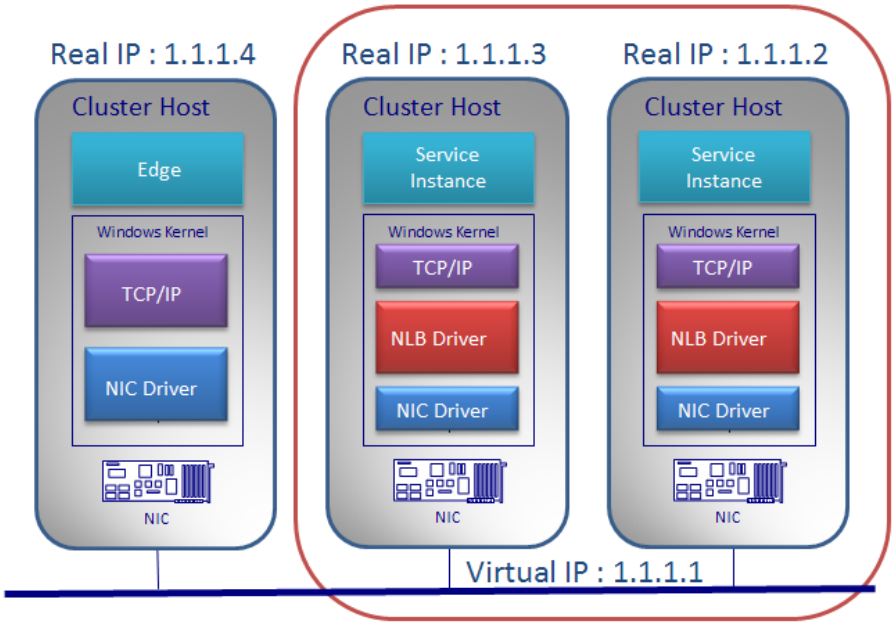


Figure 3.12 – Implementing the Service Instance Pattern using Windows NLB Cluster. The Edge is deployed outside the cluster and each Service Instance is deployed on a machine which is part of the NLB cluster.

In Figure 3.12, we see that relying on technology (Window NLB in this case) can simplify the scaling of the service. The Edge sends the requests to the virtual IP address, and the NLB cluster takes care of routing it to the appropriate service instance. The instances themselves are not aware that they are clustered. The obvious tradeoff here is, of course, that the granularity of control is weighted against ease of use, maintenance, and development costs.

The last issue in regard to the Service Instance pattern is shared state. As state earlier, it's helpful to store shared state in a shared resource such as a database. If you still need to maintain state inside each service instance, you need to looks at distributed cache solutions, such as NCache from Alachisoft or Azure cache services on the .Net platform, or GigaSpaces and VMWare's Gemfire on the JVM. Additional distributed cache option are dedicated solutions like like memcached and redis.

3.4.4 Quality Attribute Scenarios

The Service Instance pattern deals with availability. Having multiple instances of the service business logic means your service is more resilient to hardware failures as well as making sure the service will stay responsive through planned downtimes (upgrades etc). Another advantage of the Service Instance pattern is the inherent increased scalability (i.e. the ability to handle increased loads by adding hardware). As usual, Table 3.6 details a few sample scenarios:

Table 3.6 – These are the quality attributes and scenarios of the Service Instance pattern.

Quality Attribute	Practical Advantage	Sample Scenario
Availability	Hardware failure resiliency	Under normal conditions, completing service requests requires less time
Availability	Reduced system downtime	Upon a server crash, the system will remain operational
Scalability	Ability to scale-out	It would be possible to deal with increased service loads with more hardware

Although the patterns so far have approached the subject of availability, let's take a look at a pattern that addresses this head-on.

3.5 Virtual Endpoint

At the end of the day a service is a type of application that is hosted on a server somewhere. What happens when that server fails? For one, we need to take care of restarting the failed service and resume request processing. You can review the Service Monitor pattern (see chapter 4), service watchdog (the next pattern) and Transactional

Service pattern (see chapter 2) for ways to monitor services and recover from failures. The remaining issues involve service recovery time, and the failure's impact on clients.

3.5.1 The Problem

First, let's consider the service-level agreement (SLA) we need to support. In many cases, especially with mission-critical software, there is an agreement in place to ensure service availability, and to contain outages to within a specified timeframe. Therefore, we have two parameters to availability: uptime, and recovery. We need to do the following:

> **How can we provide services with location transparency, and gracefully recovery from failure, that would not affect consumers?**

If you service is truly stateless, you can scale the service using the Service Instance pattern described earlier. However, in many cases, this may not provide a completely seamless solution to the service consumer. The fact that there are multiple instances of the service may be exposed to the client. Let's explore a pattern that helps to resolve this, and improve availability.

3.5.2 The Solution

The ideal solution is to run redundant instances of the service, but have it still be accessible from the client through one address, appearing as a single instance.

> **Implement the Virtual Endpoint pattern for location transparency, which wraps multiple instances of the Edge component, thereby creating a Virtual Endpoint, or virtual Edge component.**

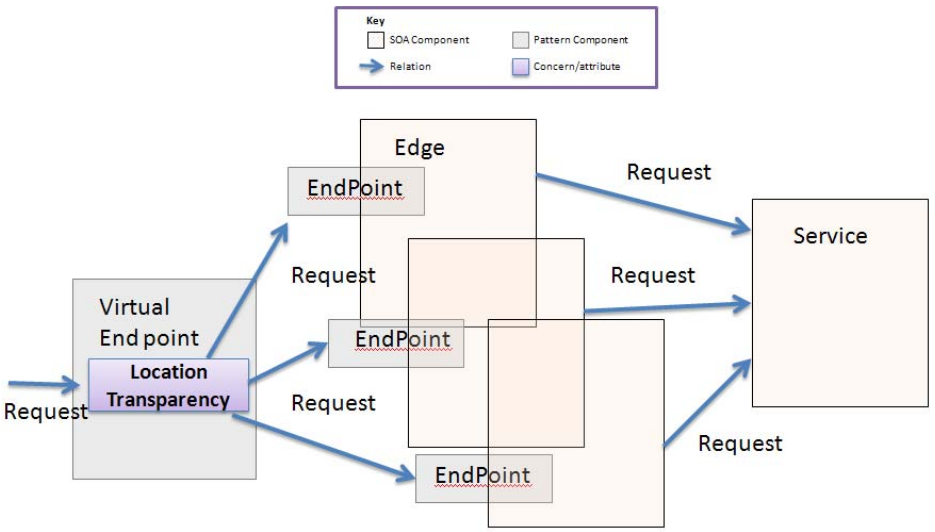


Figure 3.13 – The Virtual Endpoint exists as a known address, but the requests are actually handled by Edge Components that exists on other, internal, addresses. The Virtual Endpoint

The Virtual Endpoint pattern, as its name implies, wraps the actual service Edge component internal addresses. Requests are routed to one or more of the internal addresses where the Edge and Service exist, essentially providing location transparency for the service. There are two variations on this pattern:

1. Implement one active and one or more standby services. The standbys service would be activated only in the event of a failure. The virtual endpoint would then serve as a switch between the two.
2. Implement multiple active services. The virtual endpoint would route requests across all active service instances arbitrarily, or according to a load-balancing algorithm.

The first option is often simpler to implement, especially if issues exist surrounding multiple instances of a service running in parallel. This can include issues with resource sharing or locking, maintaining request message ordering, and so on. These issues can often be resolved through the use of enterprise software, such as a single shared database for resource issues, and a transactional queue that maintains queued message processing order.

It's usually easier to implement the Virtual Endpoint pattern in the Edge, as it's more likely to be stateless, allowing the service to maintain state independently. If you use a Service Registry it can help maintain an entry for backup service addresses.

The nice thing about the virtual endpoint patterns is that it is very simple to implement.

3.5.3 Technology Mapping

For services that are based on web-service standards, such as REST, or SOAP, the technology mapping is straightforward. For instance, you can deploy an off-the-shelf solution such as load-balancing technology . With this, you get both availability and some scalability.

If you use messaging technology rather than web technology for service interaction you can use Enterprise Service Bus (ESB) products like Mule or FuseESB (and others) to expose the virtual endpoint.

When you need smart routing to actual end-points e.g. when you have different service level agreement for different tenants you can use solutions like HA-proxy (a smart load balancer) or again ESB products.

ESBs and OS-level solutions can also help introduce virtual endpoint to stateful services via clustering in active/passive or a sharded active/active setup.

BEWARE OF THE "SPLIT BRAIN " PROBLEM

When you implement a clustering solution for availability, you need to watch out for are communication problems where different nodes in a pair don't see each other on the network. This "Split brain" phenomena occurs when more than one server claims to be the master. As a result, the servers and their data are not synchronized, which can result in partial or incorrect responses. Most clustering and high-availability products address this potential problem. However, you still need to be aware of this problem in case your solution doesn't protect against it.

The provider of the virtual endpoint (ESB, load balancer etc.) should also take care of actual end-point failure and re-route requests to an active end-point The provider can also help with state management by supporting session stickiness so that requests from the same source would get to the same handler.

3.5.4 Quality Attribute Scenarios

In its simplest form, the Virtual Endpoint pattern provides location transparency, which provides availability and scalability, but also helps with maintenance and software upgrades.

Table 3.7 Virtual Endpoint pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using the Virtual Endpoint pattern. PUD

Quality Attribute	Practical Advantage	Sample Scenario
Availability	Hardware failure resiliency	Upon a server crash the system will resume operations in 2 minutes
Maintenance	Easier upgrades	Individual service instances can be upgraded without

		disrupting service availability

Continuing with the topic of availability, which helps maintain the service when something goes wrong, it's important to know *when* something goes wrong. The *Service Monitor* pattern (discussed in chapter 4) helps here, as well as the *Service Watchdog* pattern, which we'll examine here.

3.6 Service Watchdog

Achieving availability is a multi-layered effort. We've already demonstrated the benefits of autonomous services (see for example Active Service pattern in chapter 2), and the Service Watchdog pattern will take a look at another aspect of autonomy. This pattern shows how a service can proactively identify faults and to try to heal itself when it does so.

3.6.1 The Problem

The Service Instance pattern from section 3.4, for example, demonstrates a pattern to cope with failure. The question is, is that enough? Our opinion is no, and here's why:

- Once you deal with failure within the service, the ability to cope with additional failure is probably diminished. For example, if the live server failed, and the service transitions to a standby server, there's no additional standby in case this server fails.
- The failure might be too much for the service to be able to overcome by itself. For instance, a poison message might take the redundant or standby servers down as well.

To increase the service autonomy and to increase overall availability, we need to both identify *and* repair problem, and then notify the appropriate system operator about the service's current status. Therefore, the problem statement is:

How can we increase availability, identify and resolve problems and failures that are service specific?

One option is to try to infer the state of the service from the way it looks to the outside. You periodically call the service (ping it), and if it doesn't respond within well-defined parameters (within a certain amount of time, for instance), you know the service may be down. This isn't foolproof, especially if there are redundant or standby servers involved. In that case, a problem may occur and remain masked because a standby server is available to answer our pings.

Alternatively, you can install agents on each of the service's servers. This will give you a more fine-grained view of the health of each server. You may also be able to get trend information per server, as well as warning signs to future failure potential such as with disks that are filling to capacity. However, there are some problems with this solution:

- You need to actively install software on each of the service's servers, which both

decreases the service autonomy and creates a management hassle.

- You still only get an external view of the service behavior. For instance, you cannot determine if the service is returning stale data out of cache because a network failure is preventing it from getting fresh data from an external source.
- Only the service really knows its wellness. For instance if the SLA of a service requires that there would be at least 3 instances alive (for a certain load), we had 5 nodes and one is down so we still have 4. The severity of the failure is not high. Another example can be if a process is still up but is taking more than usual. The definition of what is "usual" is something that the people that developed the service know and can be part of the service's code/configuration
- There are situations where not all of the services are under your control, and you cannot access their hardware

Yet another option is to actively contact service and actively poll them for state. This allows you to build servers that deliberately report on potential problems, and communicate trends that can lead to problems over time. For example, it can report on growing log files, falling disk capacities, network outages or external service call failures, or low-memory situations. The solution may not perfect since it is the observer's responsibility request the information and act on it. For example, if the rate at which the observer samples the service is not fast enough, it can miss vital information., However, this approach is on the right path; all we need to do is add an element of autonomy to it, as we describe in the next section.

3.6.2 The Solution

A solution where the service watches over itself is often not good enough, as you normally require a human operator to be alerted to potential trouble. The solution we propose is a combination of those we've described so far:

Implement the Service Watchdog pattern, where the service actively monitors its internal state, acts on potential trouble and tries to heal itself, and continuously publishes its status.

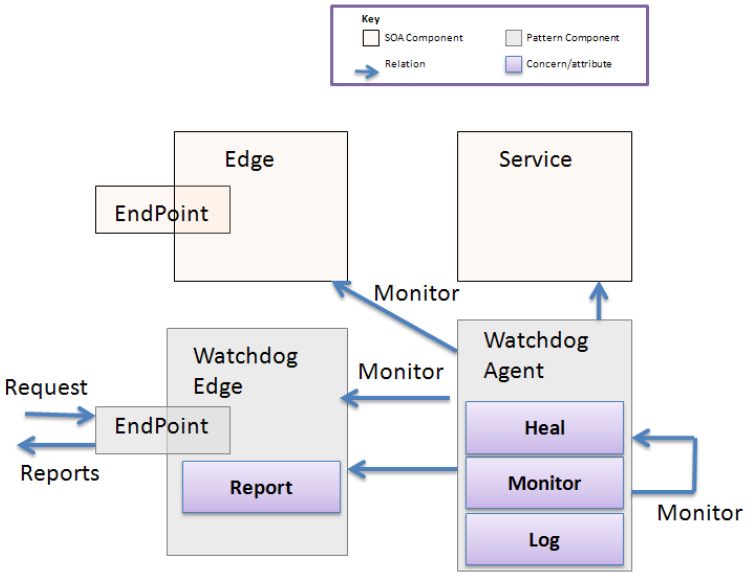


Figure 3.14 – With the Service Watchdog pattern, the Service component sends health reports and listens for requests. The watchdog component receives these reports, and tries to heal itself before the problem gets worse.

The pattern revolves around a single idea: increase the service's responsibility by combining two complementary concepts: reporting, and self-healing. The first is the *watchdog agent* concept, where the service implements the Active Service pattern (see chapter 2 for more details), and contains a component in charge of monitoring the service's state. This component publishes (see the *Inversion of communications* pattern in chapter 5) the service's state periodically, and also when something meaningful occurs. It's important to note that the fact that the service actively publishes its state doesn't have to mean it cannot also respond to inquiries regarding its health (akin to living a comment on a blog and getting a response from the author)

The second important concept in the Service Watchdog pattern is that of the *Watchdog*. This component listens for information gathered and published by the agent component, and acts on that information in a meaningful way to increase the reliability and availability of the service. While there are many ways to implement self-healing are endless, many of which are application-specific, few common examples include:

- Providing a fail-fast mechanism i.e. to stop a process when the state of the component is not certain
- restarting a failed components (e.g. as a reaction to a fail-fast),
- Implementing circuit breaker mechanism e.g. preventing a retry on a database

connection when the database is down

- Clearing junk like deleting logs, temporary files etc.

WATCHDOGS

Watchdog is a term borrowed from the embedded systems world. A watchdog is a hardware device that counts down to zero, at which point it takes action, such as to reset the device. To prevent this reset, the application has to "kick the dog" before the timer runs out. If the application doesn't reset the counter, it could means that the application has stopped responding, and a reset would fix that.

Let's discuss the advantages of the Service Watchdog pattern over the other options presented above.. The service watchdog pattern combines the benefits of an agent that actively monitors the service's health with the internal knowledge of how to maintain service continuity. For instance a service is best equipped to know it it's processing is running slower than usual. If there are many instances of the service, the service should know how many copies are really needed and how many are just for redundancy etc.

In one project, for example, we inherited a situation where there were interdependencies between processes running on different servers, as part of a single service. When the process was down on one server, the process on the second server didn't function properly, and vice versa. The end result was something like the situation in figure 3.15.

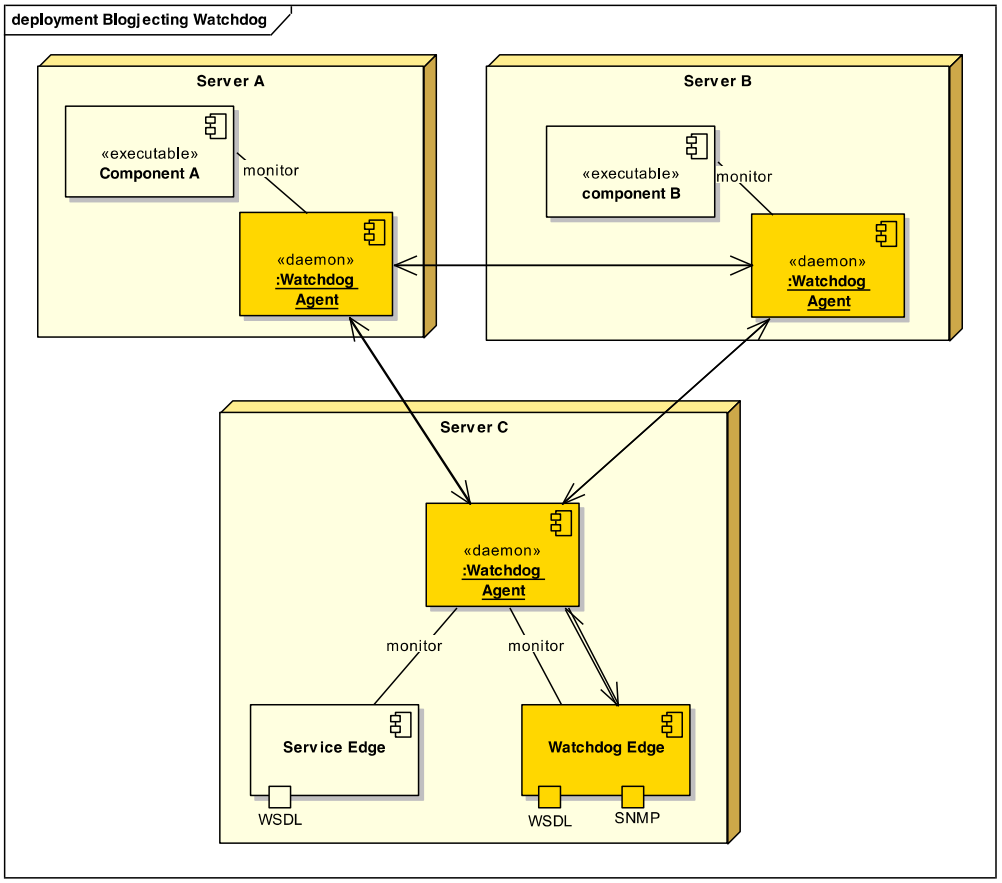


Figure 3.15 - The daemon processes on the servers monitor the running components on each server. With the Service Watchdog pattern, the Watchdog edge component exposes the current the current state through a web-service interface, and as SNMP traps

The Watchdog agent on each server node monitors the components. The agents communicate amongst themselves to examine the dependencies. The watchdog Edge component provides a web-service description language (WSDL)-based endpoint where other services can query it for the service's health. It also publishes simple network message protocol (SNMP) traps to an external SNMP monitor (e.g. HP-Openview). The simpler you keep the components, the less risk of failure. Let's take a more thorough look at the technology mapping options

3.6.3 Technology Mapping

Implementing the Service Watchdog in an enterprise will usually pre-determine the protocols you will have to use. There are many third-party monitoring packages available, such as Nagios , HP-Openview, IBM-Tivoli, or Microsoft Operations Manager. In these cases, you can use the SDK of the monitoring software, such as the CA-Unicenter Agent SDK, and so on. There are even third-party software packages to help you build agents. For example, OC Systems offers product, called Universal Agent, which you can use to write agents for CA-Unicenter. With the emergence of SOA-specific tools, such as those from Amberpoint's SOA Management System, or Weblayers' suite of products, you can implement standard WS-* based monitoring.

At the service level you can use standard mechanisms like performance counters on .NET ans JMX mbeans in Java to support emitting statistics on how well the service is doing. On one system we also configured a log listener that transmitted error and fatal log messages to the watchdog to help identify problems.

Regardless of the specific technology the important point is to let an agent that is controlled by the service determine when the service is health – the results themselves will be manifested in an external tool as noted above (and as discussed in the service monitor pattern see chapter 4).

3.6.4 Quality Attribute Scenarios

The Service Watchdog pattern is helps improve the overall reliability of the service, and allows it to maintain its autonomy. Monitoring and self-healing service can overcome minor problems, which results in better overall availability. Table 3.8 outlines some of the quality attributes that this pattern helps you achieve.

Table 3.8 – These are the architectural scenarios and benefits from applying the Service Watchdog pattern

Quality Attribute	Practical Advantage	Sample Scenario
Availability	Improved failure detection	Upon a failure or degraded performance, the system will alert the administrator (via SMS) within a well-defined amount of time.
Reliability	Increased autonomy	During normal operations, the system will clear all its temporary resources continuously

Once we begin to monitor a service and collect data, you begin to find new uses for that data. For example, you can examine trends in incoming request messages to try to locate

attacks on the service. Monitoring data can be used to analyze the service's behavior over time, predict failures, and help to increase its maintainability.

3.7 Summary

Performance, scalability, and availability are related attributes of any software system. Many times, the best was to solve a performance problem is to scale the solution. Once you do this, you often find that the same approach can be used to increase the solution's availability. This is especially true when we combine patterns, and multiply their individual quality attributes. In this chapter, we examined structural patterns to help increase performance, scalability, and availability of services in an SOA.

We covered the following patterns in this chapter:

- Decoupled Invocation – queue up requests to deal with peak loads and increase reliability.
- Parallel Pipelines – break a process to steps to increase throughput.
- Gridable Service – use grid technology for computation intensive tasks.
- Service Instance – deploy multiple instances of services to help with scalability.
- Virtual Endpoint – provide location transparency to help service availability.
- Service Watchdog – self monitoring and healing for services

The final pattern in this chapter, the Service Watchdog pattern, serves as a good introduction to the next chapter, since it introduces the topics of maintainability, and security.

3.8 Further Reading

This section provides links to resources (web or otherwise) for all the technologies discussed in this chapter.

Table 3.9 - Resources for further reading on topics covered in this chapter.

Area	Resource name/link	Why
Decoupled Invocation	LMAX Architecture (disruptor pattern)	The disruptor pattern is a pattern to create low-latency lock free queue between writers and readers.
Parallel Pipelines	Pattern Oriented Software Architecture – a system of patterns by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal	Parallel pipelines is an application of an Parallel pipeline architectural pattern described in this book

Gridable Service	Scaling SOA with Distributed Computing – Robert W. Anderson and Daniel Ciruli , Dr. Dobb's Journal 2006	It describes the same notion of adding a grid to scale SOA

4

Security & Manageability patterns

As was mentioned in chapter 1, SOA promotes loose coupling by its emphasis on interfaces, standard based contracts, and service autonomy. SOA's loose coupling of services makes it (relatively) easy to create systems by composing services together as well as updating services without disrupting the functionality of other services that interact with the changed service. Indeed SOA is a truly an open architecture style. This openness, gives a lot of benefits like agility and easier integration, but it also opens the door to many security threats and manageability challenges. In the past it was always a tradeoff choosing between openness and security or distribution and manageability so you might think it would be difficult to weave security and manageability into SOA without violating SOA's principles, however as you will see reading this chapter, a good balance between these somewhat contradicting quality attributes can be achieved.

Before we'll dive into the solutions – let's take a brief look at some of the problems they try to solve. Software systems and especially distributed and connected systems have to deal with many threats such as repudiation - someone denying that he sent a message; another example is distributed denial of service attacks which is quite common these days.

One of the actions you need to do when designing a system is threat modeling, which is, a way to understand the security requirements of the system (each identified and prioritized threat needs to have some security measures to mitigate it). In "Writing Secure Code" Michael Howard and David LeBlanc describe 6 threat types: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privilege – which form the acronym STRIDE. Table 4.1 provides a short example for each threat type.

Table 4.1 Threat categories. Software systems have many security threats. These threats can be classified into six categories that go by the acronym STRIDE.

Threat	Examples
Spoofing	man in the middle replaying message; Impersonating a consumer and sending a message in its name
Tampering	Changing the content of request or a reaction
Repudiation	A consumer sending a request and then denying it ever sent it
Information Disclosure	Exposing internal information in an error message
Denial of Service	Flooding a service with bogus requests
Elevation of Privilege	Executing a request that the consumer is not authorized to execute

I will discuss which of the STRIDE threats can be mitigated using for all the security related patterns instead of the general type scenarios ("quality attributes") section used for other patterns.

One aspect of security is obviously keeping out attackers and preventing malicious effects. Another important aspect of security is monitoring for problems and ensuring that security guidelines are followed. The monitoring facet of security is also a part of other quality attributes – manageability and governance that we'll also touch in this chapter. Another communality between security and manageability/governance is that they are also often neglected. It seems however, that organizations pursuing SOA tend to promote governance more than before. Both Security and manageability are important to ensure that a solution will be working and running as expected – security in making sure no external and unfriendly elements interrupt the service and management by ensuring everything is well on the inside.

As I already mentioned, chapter 4 contains patterns to cover both Security and Manageability. Figure 4.1 below show which of the SOA components mentioned in the SOA definition in Chapter 1, the patterns in this chapter touch:

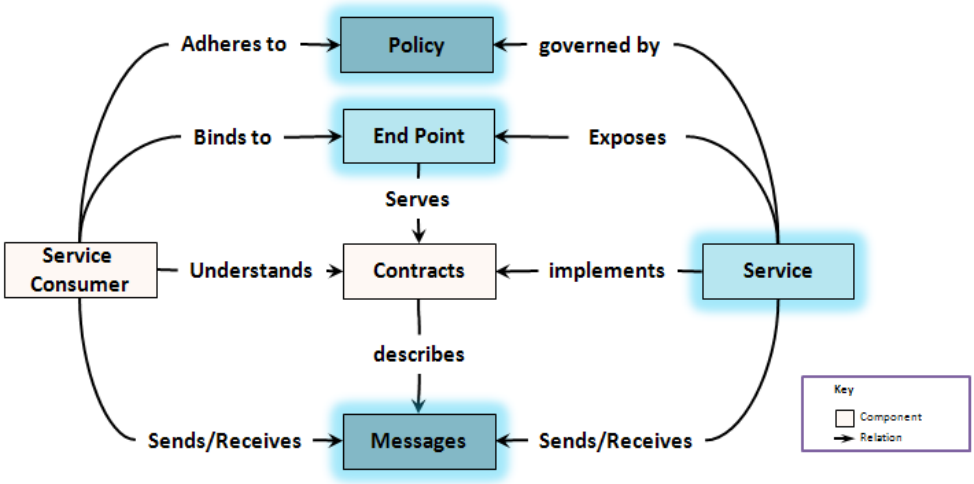


Figure 4.1 Chapter 4 focuses on security and management patterns. In previous chapters the focus was mainly on the Service itself, as we move to security and manageability the focus shifts to the interface of the service and the patterns in this chapter touch the Policy, End Point and the Messages components of SOA and not just the service

As illustrated by figure 4.1, the focus of this chapter is on the peripheral components of the service i.e. the messages, policies and the end point more than the service itself. It is better to maintain the service focus on the business functionality than to clutter it with general concerns. Dealing with security and manageability outside of the service allows the service to maintain that focus. Note that we have also seen this with the introduction of the Edge Component (see Chapter 2). This does not mean you shouldn't still write secure code or use logs when you develop the service itself. It just means that handling aspects such as authentication, authorization etc. are better handled externally.

Table 4.2 below lists the patterns discussed in this chapter and the problems that they address

Table 4.2 list security and manageability patterns covered in chapter 4.

Pattern name	Problem address
Secured Messages	How can you secure specific messages or parts of messages that are exchanged between two or more services?
Secured Infrastructure	How can you increase the overall security of message exchange between services with minimal impact on the services involved?
Service Firewall	How can you protect a service against detect malicious incoming messages and

	prevent information disclosure on outgoing messages?
Identity Provider	How can you get an efficient authorization and authentication scheme in an SOA?
Service Monitor	How can we identify problems and faults in services, attend them to ensure the overall business's availability?

The first pattern we'll discuss is the Secured Messages pattern, which is a simple pattern that talks about, well, er, securing messages.

4.1 Secured Message

The first pattern in this chapter has to do with one of the most fundamental components of SOA – the message. Messages, as was explained in chapter 1, are the components that transport data between services and their consumers. SOA based system are, by definition, open, distributed and most importantly connected, which means that we have a lot of these messages going back and forth. We can control what happens to the message inside the service and we may have some control on the service consumers – but what about the space between the services? I remember listening to a presentation by Pat Helland on messages and data in SOA where he called this space "no man's land" and in a sense this space is exactly that - especially when the messages travel over a public network such as the internet. Ok, but what does that mean?

4.1.1 The Problem

The fact that messages travel in this "no-man's-land", this space between the services, makes them prone to all sorts of threats. Many of the threats have to do with a class of attack known as "man in the middle". Figure 4.2 shows the basic template for a man-in-the-middle attack – simply put it means that when a message leaves the service or the service consumer someone lurking on the wire can take a look at the messages and tamper with them.

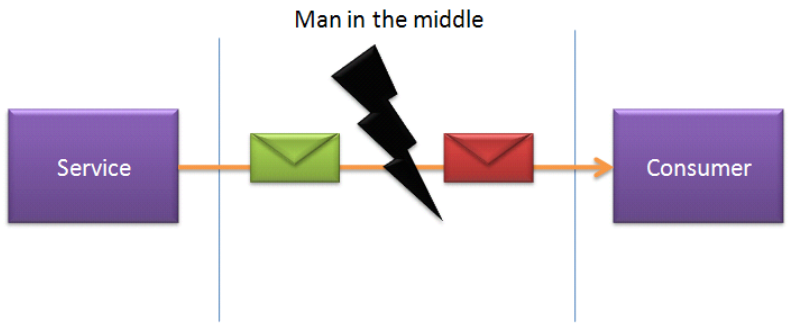


Figure 4.2 Man-in-the-middle attacks. An attacker listens in on messages that travel unprotected space and can examine or even change messages.

As I said, the scenario illustrated above is the base for several types of threats and we need to find a way to protect ourselves against them. For instance, we need to find ways to:

- Protect privacy – sometimes messages contain confidential information, or at least private. Maybe you do not want everybody to know your company's account details when you send an order message to a supplier.
- Protect integrity – you don't want anyone to change the messages you send. You don't want that 100$ order you've requested changed to 10 million dollars.
- Protect against impersonation – You don't want anyone withdrawing money from your account "on your behalf" by faking the credentials you've sent in a message.

Note that while "man-in-the-middle" scenarios are important they are not the only threats you need to handle, for instance you also need to protect against repudiation –you don't want a client of yours to deny ever sending that 10 million dollar order the minute she got your merchandize.

While the examples above are for financial transactions, the same issues are relevant for other types of messages such as sending student grades in a university system or relating fingerprints and person identities etc. or any other thing you want your services to handle. The question is then:

How can you secure specific messages or message fragments that are exchanged between two or more services?

The naïve thing to do is, well, to do nothing and hope for the best. This may sound like a stupid approach, but the fact is that I've witnessed oh too many systems where this was exactly the "solution" used. The obvious downsides are that the messages are prone to all the threats mentioned above. There may be some edge cases where it doesn't matter, but as a rule, this is probably not a good approach to take.

One option we should consider is if we can use a secure channel (see the Secured Infrastructure in section 4.2) this is a good option in the technical and architectural sense, as it takes the burden of security off of the service. The main problem with this approach is it is harder and sometimes impossible to make this work in an uncontrolled environment. There are situations where the infrastructure is limited and cannot provide the solution you need. The simplest example for that is when you only want to encrypt part of the data – say the credit card number, but want to leave the other part open. Another example for Secured Infrastructure limitation is that many times infrastructure is only suitable for point-to-point scenarios (e.g. SSL/TLS for internet security) and you may need multi-message and multi-party interactions. Furthermore, if you temporary store the messages in a cache or other less-secure temporary storage – securing just the transport may not be enough.

4.1.2 The Solution

If you can't use Secure Infrastructure or if the level of security it can provide is not good enough, what's left to do is to take care of securing the messages yourself and

Apply the Secured Message pattern to your messages and add message-level-security.

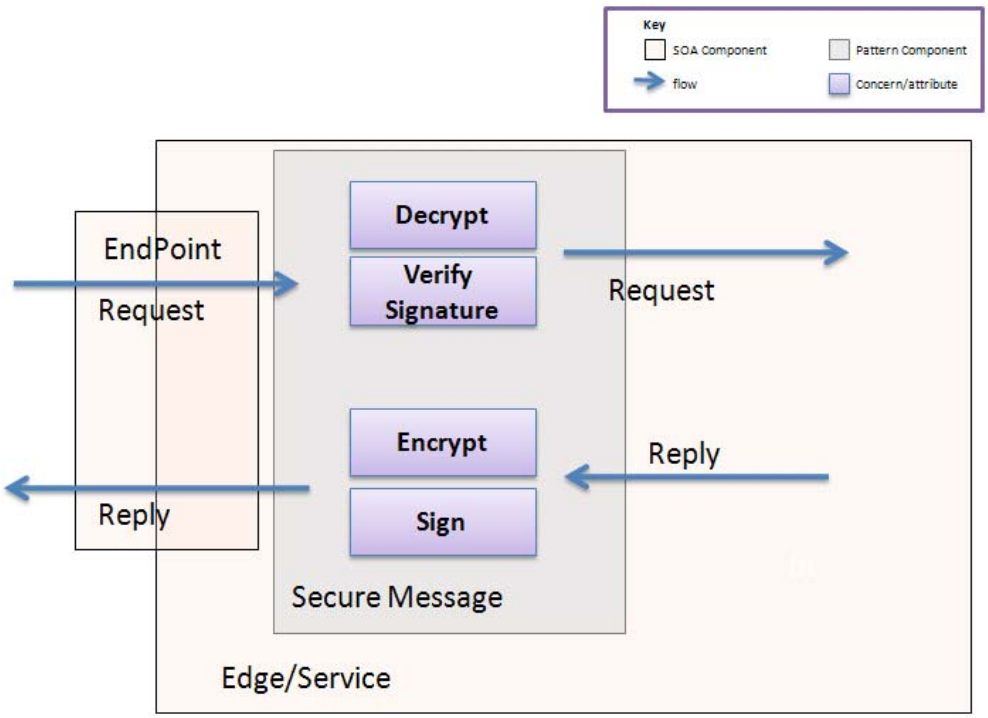


Figure 4.3 The Secured Message Pattern. The Secured Message pattern defines a single component that handles both Decryption/Encryption and Digital Signatures and their verification.

The Secured Message pattern is composed of a single component, which is responsible for enforcing the security on top of the raw-messages. The two common security capabilities for the message are

Encryption or decryption of messages. – Encryption and Decryption can help solve the privacy scenario. Since now someone looking at the message will have a hard time figuring what the message is. I used the term "hard time" rather than "will not be able to" since the

strength of the encryption depends both on the strength of the algorithm used and on how protected the key is.

Digital Signatures - Digital signatures can help solve the integrity problem. When you digitally sign a message you can get an indication if the messages was altered and it doesn't match the original that was sent. Note that digital signatures just tell you that something is wrong and not what went wrong. Digital signatures can also solve the repudiation problem as when someone digitally signs a message she needs to use her private key to sign the message which proves that she originated the message.

IDEMPOTENT MESSAGES

In addition to encryption and signing it is also worthwhile to put the time into making messages idempotent. Idemopotence is a mathematical term which basically means that calling a function multiple times does not change the result f(f(x))=f(x). Idempotence for messaging means that the messages should be constructed in a way that will enable the processing service to be immune to reprocessing a message. So that if a service receives the same message again for some reason it would be able to handle it without changing the state of the system. For instance if we consider the financial scenarios mentioned in the problem section – we wouldn't want that "withdraw 1 million dollars" message to be processed more than once

A common way to achieve idempotence in messages is to add a version number or transaction number so that you can identify. It is important to note that adding idempotence, is a choice you have to make and implement on your own as part of the service implementation. Any "infrastructure level" implementation will not be meaningful in the business context and will only handle network level retries/replays.

Sometimes it is possible to make the message idempotent regardless of the receiver e.g. "set discount to 10$ for order 123". Even if you handle this message several times the discount would still be 10$ (compare with a message that says "deduct 10$ from order 123") Sometimes we still need to make sure that the service an Idempotent Receiver(see Enterprise Integration Patterns) so that it would register which messages it already handled and check against that list before processing incoming messages.

As mentioned above the Secured Message is not a replacement for the Secured Infrastructure. The main goal for using it is to solve the scenarios that aren't handled well by Secured Infrastructure like partial encryption of data, temporary storage of data, multi-party secured sessions, and singing un-encrypted data. One reason to consider partial encryption is that impact of full encryption on latency. It takes much more time to encrypt/decrypt every message than it is to encrypt several attributes or fields in a few specific messages.

Note that even when you use the Secured Messaged pattern it doesn't mean you develop everything from scratch since if you'd do that you are likely to have insecure solutions.

Instead it is better to rely on cryptographic capabilities of your development environment you use – we'll expand on that in the technology mapping section below

4.1.3 Technology Mapping

As mentioned in the patterns structure in Chapter 1, the technology mapping section takes a brief look at what it means to implement the pattern using existing technologies as well as mention instances where current technologies implement the pattern.

The basic technology mapping for the Secured Messaging pattern is to use the cryptography libraries of your development technology you are using. Both Java and .Net have the notion of Cryptographic Service Provider, which lets you abstract away the implementation of the cryptographic algorithms. For example in one application I've architected we used this feature to seamlessly replace a software implementation of an algorithm with an HSM (Hardware Security Module) based one to accelerate encryption and decryption speeds.

Building SOAs you are more likely than not using XML for your messages (although other options like JSON or proprietary format are also possible) – if you do then two technological standards you should be aware of are XML encryption and XML signatures, both are W3C standards and both are supported by many development environment in for instance Apache XML Security project has implementations for Java, C++ and .NET 4 has specific namespace that deals with XML cryptography: System.Security.Cryptography.Xml. XML encryption allows you to encrypt specific elements within an XML for instance if we have the XML in listing 4.1. we may want to secure the Account information since it holds both an account ID and credit card data

Listing 4.1 – An unsecured XML that holds sensitive data (account ID and credit card number).

```
<Order>
     <Account>
            <AccountID>1234-6789</AccountID>
            <Payment>
                   <CardId>9999-5678-9123-4567</CardId>
                   <CVV2>123</CVV2>
                   <ValidBy>02/05/1999</ValidBy>
                   <CardName>visa</CardName>

            </Payment>
     </Account>
     <Items>
            <Item name=”Mashu”>
                   <ItemId>123-456-789</Id>
                   <Quantity>10</Quantity>
            </Item>
     </Items>
</Order>
```

If we use one of the above mentioned method and encrypt it to the standard we can get something like the XML in listing 4.2 (depending on key and algorithm)

Listing 4.2 – The XML from listing 4.1 with the account key encrypted using the XML encryption standard

```
<?xml version='1.0' ?>
<Order>
     <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
            <CipherData>

     <CipherValue>FF5BDA12C3EE3A238FCD8721AE9354</CipherValue>
            </CipherData>
     </EncryptedData>

     <<Items>
            <Item name=”Mashu”>
                   <ItemId>123-456-789</Id>
                   <Quantity>10</Quantity>
            </Item>
     </Items>

</Order>
```

Note that XML Encryption standard also defines an a tag (EncruptedKey) which is similar to the EncryptedData element in listing 4.2 to also support key exchanges in a secure manner.

Note that some threats, e.g. cross site request forgery (CSRF also known as XSRF) can be mitigated by using hashes and not encryption (it is much faster to create hashes). The Open Web Application Security Project (OWASP) suggests a “synchronizer token pattern” where for every session the server generates a unique random hash, which is hard to guess by attackers. Requests by clients should include the token so that the server know the requests are valid (see further reading and the bibliography for more details).

4.1.4 Quality Attributes

As noted in the introduction of this chapter, the quality attributes section for security patterns discusses the threats that the pattern helps prevent.

The Secured Message helps deal with a few threats derived from the presence of a “man-in-the middle.” Table 4.3 below shows the threat categories and the actions that an implementation of the Secured Message pattern can take to avoid these threats.

Table 4.3 Threat categories and actions. List of the action that implementations of the Secured Message pattern can perform and the threats these actions mitigate.

Threat	Actions
Spoofing	Verify signature using the senders public key to prevent impersonation

	Add time stamps, sequence numbers, or expiration times to messages and/or implement idempotent message to cope with replay attacks.
Tampering	Encrypted messages cannot be changed without ruining the message
Repudiation	Adding a time stamp and requiring signature on messages and adding time stamps prevents a sender from claiming he didn't send a message.
Information Disclosure	Encrypt important information (or the whole message) to prevent reading sensitive data

In regard to general quality attributes we can see that the Secured Message pattern is a little problematic. One problem with the Secured Message pattern is that is requires a lot of work on the service implementation, which means it has an impact on maintainability. Also since it is our responsibility to implement this pattern we have potential problem as we need to be careful and comply with standards otherwise it can have a bad effect on interoperability.

We can simplify some aspects of the Secured Message pattern using the Secured Infrastructure pattern – which we will discuss neduc

4.2 Secured Infrastructure

The Secured Infrastructure is relatively simple from the architecture perspective but it has a lot of details and substance in it's technology mapping. The principle behind the Secured Infrastructure patterns, as its name implies is finding a communication layer that is secured and using it for services' communications. The complication here is to decide on the appropriate technology mapping to fit your needs and once done to utilize that technology properly. The bulk of the discussion for this pattern will be in the Technology mapping section, but first let's briefly introduce the problem and the solution.

4.2.1 The Problem

To introduce the problem let's recap the scenarios presented in the Secured Message Pattern. As messages flow in the space between the services – which includes routers, networks, sometimes even public networks (such as the internet) we need to find ways to protect the messages against prying eyes and malicious onlookers. Essentially we need to protect privacy, protect the messages integrity and protect against impersonation:

- Protect privacy – sometimes messages contain confidential information, or at least private. Maybe you do not want everybody to know your company's account details when you send an order message to a supplier.
- Protect integrity – you don't want anyone to change the messages you send. You don't want that 100$ order you've requested changed to 10 million dollars.
- Protect against impersonation – You don't want anyone withdrawing money from your account "on your behalf" by faking the credentials you've sent in a message.

As we've seen basically we have the same set of problem – the added constraint, for this pattern, is that unlike the Secured Message pattern we want minimal work and impact on the services. Maybe there's a way to patch security on the outside without putting a lot of effort on the service and the developers that program it. Thus the problem is

How can you increase the overall security of message exchange between services with minimal impact on the services involved?

One option is to develop the security solution yourself, in order to try to minimize the effect on the services we can put most of the security related code in an edge component (see Edge Component pattern in chapter 2) however we still have to develop the security solution and even more importantly test it . Also we need to put special effort to make sure we use security standards to enable interoperability with external parties –don't forget that openness is an important trait for an SOA.

So what's the other option?

4.2.2 The Solution

Well, if developing a solution by yourself is not a great option the other option is to try to find a solution developed by someone else or in other words:

Apply the Secured Infrastructure pattern and use 3rd party secured solutions as the communication infrastructure for the services.

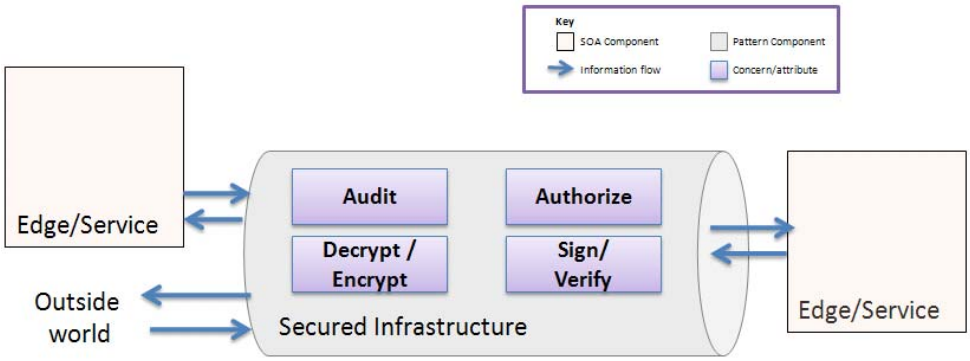


Figure 4.4 The Secured Infrastructure pattern. The idea behind this pattern is to buy (or build) a common secure communications infrastructure for the services, which is external to the services and handles all the messaging traffic for all the services.

The main idea behind the secure infrastructure is to find an off-the-shelf solution that will solve as many as the security challenges we have by configuration alone. This is a real boon since we can develop our services without thinking about security and then just change a few configuration files et voila the system is secured.

One caveat, however, is that when you apply the Secured Infrastructure pattern and decide to turn security "on", the granularity of the decision is usually limited. So that, for example, you can choose set a secure channel but then all the messages that go through it will be encrypted. The problem with that is that it can have an undesired or unplanned effect on throughput and latency of the messages. Sometimes it is a necessity since, everything in every message has to be secured but sometimes a fine grained control on security will yield both the needed level of security and better performance. One way around this problem is to add an additional unsecured channel and then you have to make sure you send the right messages on the right channels and that messages on the unsecure channel don't leak information that should be sent on the secure channel.

In order to make sure what's the right option for the solution you are building it is recommended to integrate security early and conduct performance tests for the application to assess the impact.

4.2.3 Technology Mapping

The Technology Mapping section usually covers both technologies where the pattern is used and ways to implement the pattern by yourself. However, unless you are technology vendor, the most likely path to take with the Secured Infrastructure pattern is to pick off-the-shelf solutions. We will cover the most common technological options: SSL/TLS, WS-Security and using ESBs.

SSL/TLS

The first, and probably most approachable option, is to use SSL (Secured Socket Layer) or TLS (Transport Layer Security) these are standard internet protocols; all web browsers support them; and they are in wide use today. SSL/TLS is the natural selection for securing RESTful services since REST builds on HTTP as it is. Nevertheless, note that that they are also supported by web services based on WS-* standards such as WCF or JAX-WS. Adding SSL support for a web-service simply involved marking it in the WSDL that describes the service's contract. For example let's say we have a definition such as the one in Code listing 4.3 which shows the skeleton of a definition of a web-service exposed as a Servlet endpoint.

Listing 4.3 - JAX-WS definition of a web service exposed as a Servlet endpoint

```
package servletws;

import javax.annotation.Resource;
import javax.jws.WebService;
import javax.xml.ws.WebServiceContext;

@WebService
public class OrderServlet {
```

```
        @Resource WebServiceContext wsContext;

        public String PlaceOrder(OrderMessage msg) {

              .
              .
              .
        }
    }
```

In order to make this web service use SSL all we need to do is add a few tags into the WSDL such as < transport-guarantee> CONFIDENTIAL </transport-guarantee> and <auth-method> CLIENT-CERT </auth-method> and we are all set. Listing 4.4 shows an except from the WSDL that configure the OrderServlet in listing 4.4 to use SSL.

Listing 4.4 – a WSDL expcert for the service in listing 4.3 that configures it to use SSL

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Secure Area</web-resource-name>
    <url-pattern>/OrderServletService/OrderServlet
    </url-pattern>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    </role-name>EMPLOYEE</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>certificate</realm-name>
</login-config>
```

By the way, If you one of those that need to understand how things work in detail you may want to check out Open SSL - which is an open source implementation of the SSL and TLS protocols.

The downside of using SSL/TLS is that they provide transport level security and that they are tied to a specific transport (http). Thus, if you are building web-services that have to use multiple transports like JMS/MSMQ and then HTTP you can't do that with SSL/TLS. Another point is that message won't be encrypted as it passes all the layers between the transport and the process on the server which might be a security risk in some situations. Lastly since SSL operate at the transport level, which means it is an all-or-nothing protocol – all the messages that flow on the channel will be secured which might be an overkill and a performance bottleneck.

As an aside note that you can use IPSec, which is even lower level technology (compared with SSL/TLS) to implement the Secured Infrastructure pattern. IPSec sits in the network level, the IP level and is completely external to the services. Essentially it allows secure communications between two hardware nodes. It isn't as versatile as SSL/TLS but it can be

used to secure the communication of a closed group of services in an efficient way and with a relatively little hassle (save for configuration and setup). IPSec suffers from the same limitations SSL/TLS has.

WS-SECURITY

Another technology mapping we for the Secured Infrastructure pattern is WS-Security. WS-Security alleviates some of the problems mentioned above, as it is a message level protocol and not a transport level one. For instance, you can choose which messages to secure and which to leave open or you don't have to encrypt a message when you just want to sign it etc. WS-Security is a WS-* standard that provides means to encrypt, sign and authenticate messages exchanged between services and their consumers. As illustrated in figure 4.5 below WS-Security adds security tokens and signatures to the message header.

Figure 4.5. The figure is an illustration of the structure of SOAP message using the WS-Security protocol . WS-Security adds a security header to the header of the message where a sender can store its security token as well as a digital signature. Additionally the sender can decide to encrypt the content of the message body (SOAP body).

The signatures are used to insure the message hasn't changed (integrity) and to verify the sender. As for Security tokens, well, the OASIS standard explains that "Security tokens

assert claims and can be used to assert the binding between authentication secrets or keys and security identities." Or in plain English a Security Token is the credentials used for authentication, authorization or both for example it can be an X.509 certificate or a user name that can carry with it a set of statements or claims as the standard calls them. Claims can be anything the sender (or someone on his behalf) care to say about the sender for instance, a claim can be the key that should be used to decrypt the message, the identity of the sender etc.

One advantage of this approach is that if you trust the security token you can immediately process the message you received while for SSL for example you need to first establish a session (exchange keys between the parties). While I mention SSL, it is important to note that WS-Security can use SSL and in fact the sample in code listings 4.3 and 4.4 does just that. Regarding trust - one option to gain that "trust" can be from having some trusted authority sign the security token – you can read more about this in the Identity Provider pattern later this chapter.

One limitation of WS-Security vs. SSL and other options is that it can only be used with WS-* web services (SOAP based). This is a limitation since while SOAP based web-services are a popular option for implementing SOA it is not the only option. Also if you use WS-Security for all the messages it is likely to be slower than SSL since SSL can work at the bit level and be streamed while WS-Security requires complete messages.

For REST based systems we have similar approach regarding authorization call OAuth (current version is OAuth2) which also uses tokens for authorization. Oauth can be used on top of the SSL/TLS approach mentioned in the previous section.

ESBs

The third technology option for implementing the Secured Infrastructure pattern is using an Enterprise Service Bus (ESB). ESBs are different from the previous two technology options discussed here (SSL and WS-Security) as ESBs are a higher-level solution. In a nutshell ESBs are integrated standards based service communications infrastructures that provide several feature like messaging, mediation and managment (see Service Bus pattern in chapter 7 for a more thorough discussion). What's important for this discussion is that ESBs offer secured communications as well as provide means to expose services using the above mentioned technologies. Additionally you can use ESBs to route messages which makes it easy to introduce additional security mechanisms such as the implementations of Service Firewall or Service Monitor pattern (see later this chapter for both). Essentially if you expose all your services over an ESB you can use it as a central point to perform the 3 A's – Authentication, Authorization and Auditing.

4.2.4 Quality Attributes

As noted in the beginning of this chapter, the quality attributes section for security patterns discusses the threats that the pattern helps prevent.

The Secured Infrastructure pattern helps mitigate threats related to 3rd party interception or inspection of messages (man-in the middle). Table 4.4 below shows the threat categories and the mitigations that are relevant to the Secured Infrastructure pattern..

Table 4.4 List of the action that implementations of the Secured Message pattern can perform and the threats these actions mitigate.

Threat	Actions
Spoofing	Verify signature using the senders public key to prevent impersonation
	Add time stamps to messages or implement idempotent message to cope with replay attacks
Tampering	Encrypted messages cannot be changed without ruining the message
Repudiation	Adding a time stamp and requiring signature on messages and adding time stamps prevents a sender from claiming he didn't send a message
Information Disclosure	Encrypt important information (or the whole message) to prevent reading sensitive data

Secured infrastructure helps protect the channel against an external attacker when the two parties involved in the message exchange are valid. Id doesn't cover malicious consumers that try to attack our service. For that we can look at another pattern – the Service Firewall pattern.

4.3 Service Firewall

In the previous patterns we mentioned that messages travel in "no-man's land". You can use the Secured Message or the Secured Infrastructure patterns to protect the messages while they travel that space - but what do we do if the sender is malicious? When an attacker send us a malicious messages (e.g. with a virus as a SOAP attachment) it wouldn't really help us that we managed to ensure that this malicious message gets to us intact and without anyone else seeing it?

4.3.1 The Problem

To illustrate the type of attacks a malicious sender can cause let's look at one of them a little closer. Figure 4.6 below an XML Denial of Service (XDoS) attack. In this type of attack a malicious sender attaches a lot of digital signatures to a message. Parsers that aren't ready for this type of attack examine each of these signatures causing the service to slow down under the load.

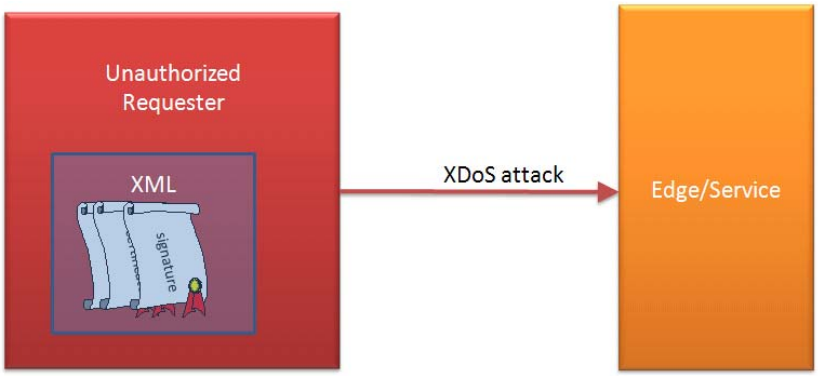


Figure 4.6: Illustration of a XDoS attack. A malicious sender prepares an XML that looks valid but is loaded with a lot of digital signatures. An unsuspecting parser will try to verify each of these signatures while hogging down CPU cycles which can result in unavailability of the service.

Another, common attack scenario is XPath injection or even plain old SQL injection where the parameters passed within a message are malicious an aim to disclose information or perform harmful operations on data within the service.

Attacks using incoming messages as discussed above are one of types of threats we need to handle, another related type of threats or problems has to do with outgoing messages. Here we need to make sure that private or classified information does not leak outside of the service. In this type of scenarios we want to find a way to make sure they hold only information allowed in the contract flows out of the service.

How can you protect a service against detect malicious incoming messages and prevent information disclosure on outgoing messages?

One option for dealing with malicious senders is to apply the Secured Infrastructure pattern (mentioned earlier in this chapter) and require certificates for authorizing clients. This means that clients that do not have a certificate will not be allowed to contact the systems. One problem with this approach is that it is only good when the number of service consumers is controlled and not open for the general public (e.g. exposed on the internet). Another limitation of the certificate approach is that it doesn't handle attacks by insiders since they are authorized to access the system.

Another option is to incorporate the security logic that screens malicious content as part of the business logic. There are several problems with this approach one is that you get code duplication (violation of Single Responsibility Principle), as there are many threats that are common to all services. Another problem is that the business logic gets tainted with the security logic which makes it both harder to write and harder to maintain.

The better option is to externalize the security to another component. Let's look at this option more closely.

4.3.2 The Solution

SOA messages are application level components – however. the notion of messages is not new or unique to SOA. We (the computer industry) already have a lot of experience with messages on a lower level of the OSI stack - the network level, specifically TCP packets and UDP datagrams. TCP and UDP have few similarities with SOA messages, the interesting ones, for the purpose of this pattern anyway, are the threats they face. Since the threats are similar maybe we can also use the same solutions we've found to work for TCP and apply them for our SOA messages.

Implement the Service Firewall pattern and intercept incoming and outgoing messages and inspect them in a dedicated software component or hardware.

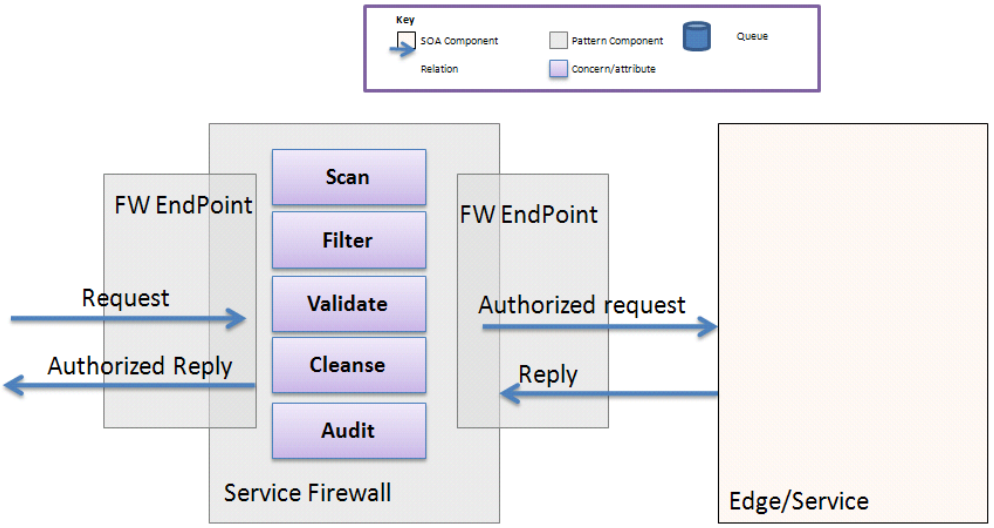


Figure 4.7 The Service Firewall Pattern. The Service Firewall sits between the outside world and the actual service (or Edge). The Service Firewall scans, validate and audit both incoming and outgoing messages. Once a message is identified as problematic it can either be filtered out or cleansed.

The Service Firewall pattern is an application of the Edge Component pattern (see Chapter 2). Figure 4.7 above illustrates how the Service Firewall operates. First it intercepts each incoming and outgoing message and inspects it. Once intercepted the Service firewall can scan the message for malicious content such as viruses or XDOS attacks and injection

attacks as mentioned in the above – Note that the firewall doesn't perform some magic that lets it deal with these threats. The point is that it is built to cope with the threats, identifies the patterns that mark a message as harmful and that it screens incoming and outgoing messages. Additionally, the Service Firewall can validate messages by making sure they conform to the contract, verifying property types and sizes etc. When a message is identified as problematic the Service Firewall can audit and log the message and then decide whether to filter it out or cleanse the problematic content and let it through.

The Service Firewall acts as a first line of defense for the service. As illustrated in Figure 4.8 below, when a request arrives at the firewall it is scanned and verified, requests that were authorized are then routed to the real service (or another Edge Component)

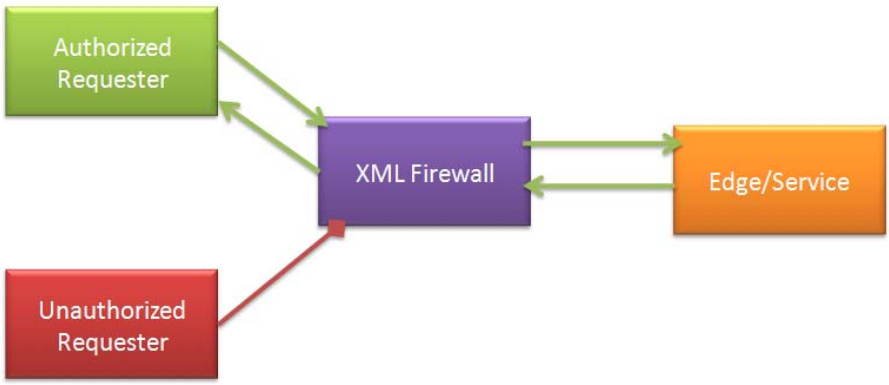


Figure 4.8 When a request arrives at a Service Firewall (an XML firewall in this illustration) it is screened for validity, for instance they firewall can check that an XML matches the predefined XSD. Authorized requests get through and unauthorized requests are rejected.

The idea behind a Service firewall is simple, the implementation, however is a little more complicated since there is a lot of functionality that has to be implemented to get each of the roles (scan, validate, filter etc.). On top of that you need a way to make sure the Service Firewall gets all the messages to be able to work

4.3.3 Technology Mapping

As mentioned in the patterns structure in Chapter 1, the technology mapping section takes a brief look at what does it mean to implement the pattern using existing technologies as well as mention instances where current technologies implement the pattern.

The simplest way to implement the Service Firewall pattern is to create a designated Edge Component where you would implement the inspection and validation logic. Once the firewall logic is done you deploy it on the DMZ. Deploy the real service behind a regular

firewall and you are all set. The Edge component will block unwanted requests that play "nice" and the regular firewall will block the other attacks.

Implementing the Service Firewall pattern without using a regular firewall is a little more problematic as an attacker can just call the endpoints that are used by the actual service and bypass the Service Firewall altogether. In these situations you can rely on the interception capabilities of the technology you use. For instance, Figure 4.9 below shows the relevant extension points offered by Windows Communications Foundation for intercepting incoming messages.

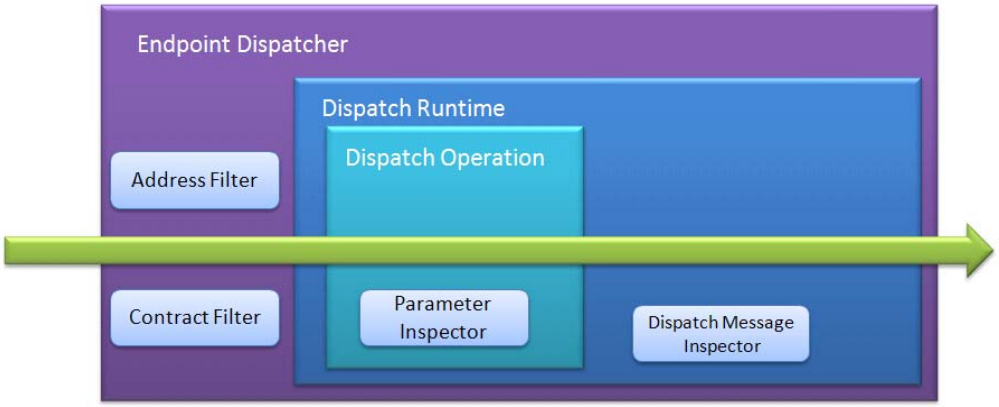


Figure 4.9. WCF supports a few dozens of extension points to control the way a message is handled when in enters or leave the service. You can use four of these extension points (depicted above) to implement the different roles defined in the Service Firewall pattern.

As illustrated in Figure 4.9 there are four relevant extension point (out of the few dozens of extension points supported by WCF) where you use classes to perform the various roles of the Service Firewall pattern. You can have classes that and verify addresses, verify contract, inspect the messages and inspect parameters – both for incoming and outgoing messages. The code snippet in listing 4.5 below defines a new WCF web-service endpoint in code and sets up a custom ServiceauthorizationManager that will be the Service Firewall instance.

Listing 4.5. a WCF code snippet to define an endpoint and add an interceptor that will get incoming messages using WCF AuthorizationManager extension point

```
var testServer = new Tester();
 var service1 = new ServiceHost(testServer, new
Uri(string.Format("http://localhost:{0}", TestServerPort)));

 var ep = service1.AddServiceEndpoint(typeof(TestingContract), binding,
```

```
string.Format("http://localhost:{0}/S1", TestServerPort));
 ep.Behaviors.Add(new WebHttpBehavior());
 service1.Authorization.ServiceAuthorizationManager = new
ServiceFirewall(); // set up an interception point for our Service Firewal
 var cp = service1.AddServiceEndpoint(typeof(ImContract), binding,
string.Format("http://localhost:{0}/Control", TestServerPort));
 cp.Behaviors.Add(new WebHttpBehavior());
```

Once we have an interception point we can define the class that would do the actual scanning of incoming messages

Listing 4.6 .NET skeleton code to perform message validation on intercepted messages

```
public class ServiceFirewall :ServiceAuthorizationManager
 {
     public override bool CheckAccess(OperationContext operationContext,
ref Message message)
     {
         var autorized= base.CheckAccess(operationContext, ref message);
         var buffer = message.CreateBufferedCopy(Int32.MaxValue);
         message = buffer.CreateMessage();
         var testMessage = buffer.CreateMessage();
         .
         .  // code to validate messages goes here
         .
         return autorized;
     }

 }
```

Another implementation option for the Service Firewall pattern is using hardware or embedded appliances. For instance, there are several companies like Layer7, IBM (Datapower appliance), Vordel and a few others that produce XML firewall appliances. The advantage of using XML appliances is that you can deploy them along with your other firewalls in the DMZ and have them serve as the first line of defense. Another advantage is that these are platforms optimized for XML handling so the performance impact of the appliances is lower vs. self coded solution. One disadvantage of using hardware XML firewalls is setup costs (tens of thousands per unit) another disadvantage is increased maintenance complexity which comes both from managing an additional hardware type and more so from the double management of your SOA contract (both in the service and in the appliance).

Whether you use an appliance or implement the Service Firewall pattern in code it can really boost the security of your services by helping prevent threats like denial of service attacks or even just save some validation efforts for the service itself

4.3.4 Quality Attributes

As noted in the beginning of this chapter, the quality attributes section for security patterns discusses the threats that the pattern helps prevent.

The Service Firewall pattern is a very versatile pattern and it can be made to handle many types of threats. Table 4.4 below shows the threat categories and the actions that an

implementation of the Service Firewall pattern can take to protect against threats in this category.

Table 4.4 Threat categories and actions. List of the action that implementations of the Service Firewall can take and the threats these actions mitigate.

Threat	Actions
Tampering	Verifying signatures and to make sure no one changed the content of request or a reaction
	Validating that messages are not mal-formed
Information Disclosure	Scanning outgoing messages for sensitive content
	Restricting reply addresses to a closed groups
	Inspecting incoming messages for XPath and SQL injections attacks
Denial of Service	Preventing XDoS attacks by examining XMLs before validating each signature
	Blocking known attackers
	Restricting Requestors addresses to a closed group
	Scanning attachments for viruses
Elevation of Privilege	Examining an incoming message for injection attacks
	Examining an incoming message for buffer overruns by validating contract and sizes of elements

In Addition to the specifics of the threats that the Service Firewall pattern helps mitigate, we can also look at it from the wider scope of quality attributes. Like most of the patterns in chapter 4, the Service Firewall pattern is a security pattern. It is interesting to note that unlike most other security patterns it is relatively easy to add it on towards the end of a project. You should note however that it is not a completely free ride and for instance you still have to measure its impact on system performance, it can add an overhead in regards to contract maintenance etc.

As we get nearer to the end of this chapter it is about time to start talking about manageability patterns. The next pattern - the Identity Provider pattern, helps make this transition as it has both security and manageability aspects.

4.4 Identity Provider

When you move an enterprise to SOA or even if you only build a single system based on SOA concepts you are likely to end up with quite a few services – and quite a few more service interactions. From the security perspective we want to make sure each of these

interactions is both authenticated and authorized. This means that each of our services has to take care of this authentication and authorization Ay, there's the rub, this proliferation of authentication and authorization raises several challenges in regard to maintenance, management, performance and security. Let's look at a sample scenario.

4.4.1 The Problem

You may remember the Subscription agency we met in chapter 2 (Active Service pattern), let's take another look at them. As a journal subscription agency, one of the more important services is the one that deals with the customer. Almost any other service in the system needs information from that service. Figure 4.10 below shows 4 simple examples – The service that deals with promotions want to get addresses, the proposals needs discount rates, and the billing, ordering need both addresses and discount rates.

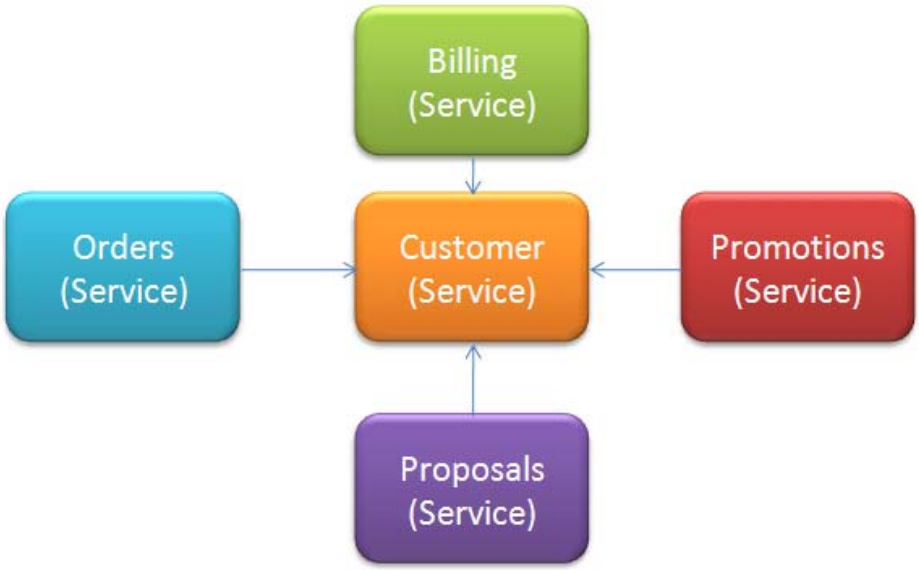


Figure 4.10 : Services interact with each other all the time. In this example we see that the Customer service gets requests from 4 different services and it has to authenticate and authorize them on every call!

So what's the problem with that? Well, as a matter of fact there are plenty of problems:

- The Customer Service needs to authenticate we are talking to an internal friendly service for each of the services that connects to it. However we don't want it to know about each of these services. After all we don't want to update the Customer service every time we add a new service this point-to-point integration is something we

wanted to avoid when we went down the SOA path.

- When we have a human in the loop too and we need to make sure she is authorized to get customer's data. For example when a user works with a UI that works with the ordering service, she might be authorized to get a customer's email (to send a an order confirmation) but not his home phone number.
- We don't want each service to "know" all the users
- If the customer service has to authorize and authenticate every call it has to spend a lot of time doing so – which adds both latency increases temporal coupling
- How do we manage the authorizations and authentications within the SOA fabric? Let's say we just added the Proposals service, how do we make the customer and any other service to know it is ok to talk to it? How do we do that for new users
- How do we revoke the credential for an employee that left the company on all the services?
- And what happens if one or more of the services is external? For instance we may have a 3rd party that handles the promotions for us, we want to let them have few details as possible from the customer's service but we do want to allow them to talk with us.

What we basically need is an efficient and secure way to handle authentication and authorization in a reality of a federated and distributed system. We need an answer to the following:

How can you get an efficient authorization and authentication scheme in an SOA?

The first question that comes to mind is wouldn't a Secured Infrastructure pattern (see earlier this chapter) solve this? Well, no, since a Secured Infrastructure takes care of the channel but how do we know we can talk with someone on that channel? We can communicate over a Secured Infrastructure to establish the identity, but we need something more.

The naive option of trying to for each service on its own is a maintenance nightmare as we need to do that work for each service and we also run the risk of introducing coupling and point-to-point integration for each new service consumer we introduce.

Writing this code once and reusing it (e.g. with an Edge Component see chapter 2) will only work if you or your team owns all the services. Also you still have a management and maintenance problem as each running instance has to be updated when a new service consumer is introduced.

Introducing an external party to handle the authorization and authentication is a step in the right direction as we can have central management of who is authorized to do what. But we still have to solve a few issues. One is that most SOA implementations are sessionless so

we need to make sure this external party won't become a performance bottleneck when each and every request has to be authenticated and authorized with it .

The second is that we don't want to couple our services to this external party but each service does need to know somehow it is talking to the right external party and not some malicious impersonator.

4.4.2 The Solution

To take the "external party in charge of authentication and authorization" to the next level we need to:

Implement the Identity Provider pattern to get Single Sign On for service consumer's authorization

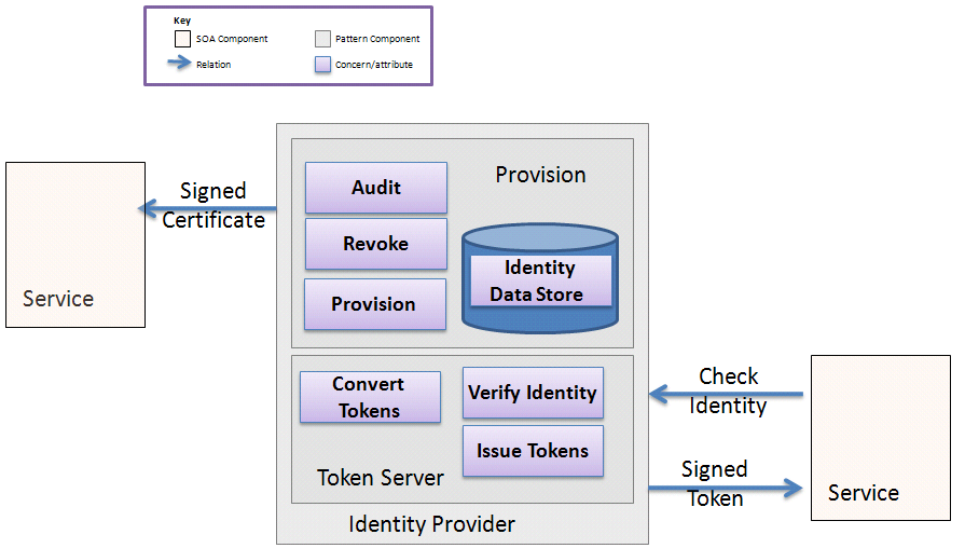


Figure 4.11: The Identity Provider pattern. The Identify Provider has two main components. One that manages the identities (provisioning) and another that is in charge of authentication- the Token Server. When a service wants to validate an identity it passes a request to the Identity Provider which returns a signed token to the service which verifies the identity. If the service trusts the Identity Provider it can also trust the verified identity.

The Identity Provider pattern is an evolution of a central identity repository mentioned above. Before I'll explain how the patterns solves the problems left unanswered by the other let's explore and understand the components of the pattern and their respective roles. The Identity Provider pattern is composed of two major components, provisioning and token server.

- Provisioning – this component is responsible for creating identities, privilege levels etc. storing these identities and supplying it to services. The provisioning component is also responsible for revoking credentials when needed. Lastly, the provisioning component can also audit and save any "identity" created updated or revoked.
- Token Server. The token server is responsible for verifying claims for identities or privileges and providing the proof that these claims are correct. It is also responsible to convert token format. Format conversion is needed as different services ,especially if they belong to different organizations, don't necessarily understand the same tokens. For instance the customer service in the example above can use x.509 certificates and the promotions service which may belong to an external PR agency might use SAML assertions (more on that in the Technology mapping section). In these cases the STS can convert between the formats while maintaining the verified identity.

How does the Identity Provider work? The core concept is trust and the mechanics for using it build on previously successful infrastructures like PKI. A service consumer, which went through provisioning with an Identity Provider sometime in the past, tries to access a service. When sending a message, the service consumer makes some assertions on its identity and/or its capabilities. If the service trusts the Identity Provider it can ask it if the claims made by the service consumer are genuine and is they are accept the service consumer's request Think about the lumberjack and the choir in the "Lumberjack song" by Monty python. The lumberjack has two assertions: "I am a lumberjack and I'm Ok.." and the choir acting as an Identity Provider confirms that "He's a lumberjack and he's OK".

If we return to the subscription agency scenario presented in the problem description the authentication and authorization can follow the steps in Figure 4.12. The Proposal service gets ready to send a request to the customer service to get a list of discounts for a customer. It will then digitally sign this request as a proof that it has credentials in the identity provider. The customer service can then check this claim with the Identity Provider, which will return to it a token or a certificate that verifies that the Proposal service is entitled for this service. The Identity provider signs this certificate with its private key. The Customer service can verify that the Identity provider signs the certificate and then since it trusts the Identity provider it can honor the certificate and return the list of discounts to the proposal service.

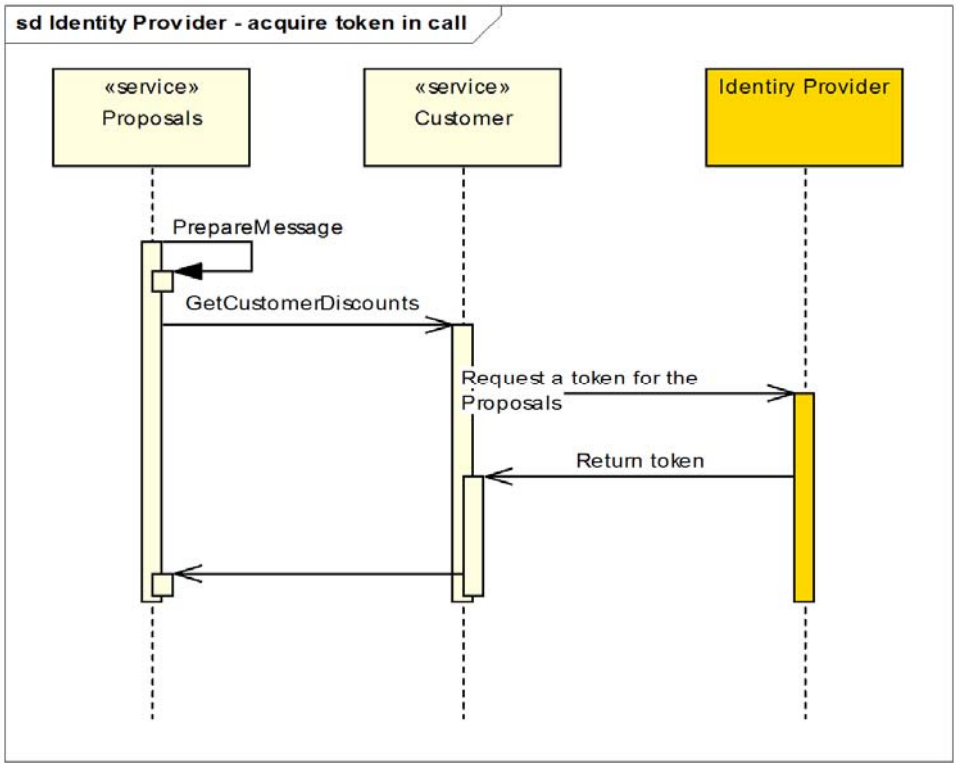


Figure 4.12 acquiring a security token. The proposal sends a request and signs it with it private key, the customer service checks the proposal's credentials against the Identity Provider, which returns certificate for the proposals service which the is signed by the Identity Provider. The Customer service, which trusts the Identity provider can then process the request of the proposal's service.

The identity Provider is an external party and thus services, like the customer service and other services don't have to figure how to authenticate callers. The process as described above also solves the coupling problem by only requiring the customer service to know the identity Provider's private key and to trust it – it isn't tied to a specific implementation of that provider.

One problem is how to prevent the Identity provider from becoming a bottleneck. One possibility is to use tokens that don't expire immediately and then have the services cache it for the next calls. Another option to do this is to pre-issue tokens during idle or low-traffic times so as to prevent flooding the identity provider in load times.. Figure 4.13 illustrate how pre-issued tokens would work.

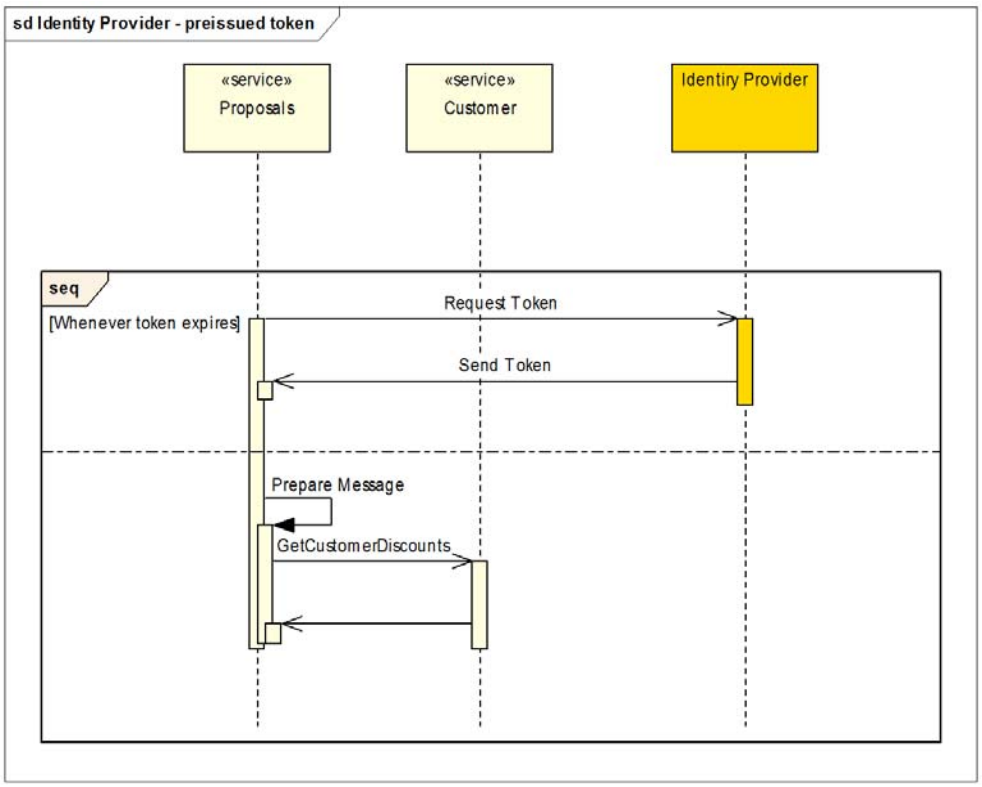


Figure 4.13 working with a pre-issued token the customer service can just process the call providing the assertions made by the proposal service were signed by a trusted Identity Providers

Now the proposals service requests a token from the identity provider and caches the signed token. Whenever the proposals service wants something from other services (and as long as the token is valid) the proposals service just sends the token along with the request. The proposals service still has to sign the request to make sure no one uses his token

The Identity Provider can be used together with the Secured Message or Secured Infrastructure patterns (see earlier this chapter) to ensure communications between the services and the Identity Provider are secured as well. Additionally it can be beneficial to use an Active Service (see chapter 2) to proactively make sure a service has a valid token – either cached or in the Identity Provider.

The identify provider pattern takes care of authentication as the distributed nature of SOA promotes the need for federated identity. A security solution will most likely require additional components that are not SOA specific like access management component /

entitlement component where you can set authorization policies, policy enforcement points to enforce these policies, an identity repository (most likely LDAP directory of some sort)

It is important to note that we need both a secured implementation of the Identity Provider pattern and a secured protocol and format to pass both credentials and assertions back and forth. The next section discusses these issues in more depth.

4.4.3 Technology Mapping

The solution section mentioned that the way the pattern works resembles the way PKI works.. This is not a coincidence since PKI infrastructures are both proven and successful. As always you can choose to implement the Identity Provider pattern by yourself however building a scalable and secure server that would also allow cross-enterprise single sign-on scenarios is not an easy feat. I would only recommend going down this path if you have specific needs and don't need to cater to the general case (such as cross enterprise single sign-on mentioned above). Luckily there are quite a few solutions that implement this pattern for you such as Shibboleth which is an open source implementation by Internet2, Oracle Identity Server, IBM Tivoli Access Manager, Ping Identity PingTrust and several others. The Identity data store can be internal to the product but can also reside on LDAP or active directory.

It is important to note that it isn't enough to have a secure sever for provisioning and token management you also need a secure tokens and a protocol for the communication of the identity information. If you don't use a secure protocol then an impersonator can assume a token which is destined for an authorized party and use it to launch attacks or acquire sensitive information . There are many ways to transport security tokens the most common ones are X.509 certificates, Kerberos ticket and SAML. X.509 certificates are more worthwhile to keep as they are relatively long lived (vs. Kerberos tickets for example). However The more interesting technology is SAML (Security Assertions Markup Language) now in version 2.0 which is much more than a security token. SAML is also a protocol for requesting and transmitting identity information.

The basic building block of SAML is the assertions, which are comprised of statements such as authentication statements and Attribute statement. Authentication statements contain the information that a requestor was authenticated and the authentication methods used to do that authentication. Attribute statements are the basis for authorization and contain information on roles, groups and any other information that exists in the identity data store.

The last part of the puzzle is a protocol to support the "Convert Protocol" capability of the Identity Provider pattern. This is supported by another WS- * protocol which is called WS-Trust. WS-Trust allows a service consumer to request an Identity Provider to exchange one token it already has to another format. As mentioned above a specific service within an SOA may understand a certain type of protocol, using WS-Trust a service consumer which wants to talk to a service that accepts tokens it doesn't have can request an Identity Provider to

replace its token. Listing 4.7 shows a request to exchange a x.509 certificate for a SAML token

Listing 4.7 SOAP body of a request to exchange a token from one format to another.

```
<wstrust:RequestSecurityToken>
   <wstrust:TokenType>SAML</TokenType>
   <wstrust:RequestType>ReqExchange</RequestType>
     <wstrust:OnBehalfOf>
            <ws:BinarySecurityToken id="originaltoken" ValueType="X.509>
             sdfOIDFKLSoidefsdflk …
     </ws:BinarySecurityToken>
   </wstrust:OnBehalfOf>
</wstrust:RequestSecurityToken>
```

The identity Provider would authenticate the request is genuine, the most common way to do that is to send this request as a WS-Security signed or encrypted request (see Secured Infrastructure pattern earlier this chapter). And if the credentials are ok it will produce a matching SAML token.

These are a lot of protocols and technologies and utilizing them all is not easy – the next section reminds us why is it worth going through all this trouble.

4.4.4 Quality Attributes

The quality attribute scenarios section discusses the architectural benefits of utilizing patterns from the requirements perspective. As was mentioned in chapter 1, most of the architectural requirement are derived and described from the perspective of quality attributes (scalability, flexibility, performance etc.). The best – through the use of scenarios where these attributes are manifested. The scenarios can also be used as reference for situations where the pattern is applicable.

The main reason to employ the Identity Provider pattern is more of a management and maintainability pattern than it is a security one, or more precisely it is management and maintenance of security related aspects. By relying on trust and certificated, the Identity Provider also enable to solve some of the latency issues usually related to adding a security layer.

Table 4.5 below shows a few scenarios where it is beneficial to use Service Monitor pattern.

Table 4.5 Identity Provider pattern quality attributes scenarios. The architectural scenarios that can make us think about using the Edge Component pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Maintainability	Adding Service	Configuring the security for a new service will take less than

		half a days work of a single developer
Performance	Latency	The cost of authenticating all request will not exceed 100 msec
Security	SSO	The system should support single sign-on for all service and human interactions
Security	Authentication	During normal operations A revoked right will be updated in the system within 5 minutes
Security	Federated Identity	Under normal operations the system should be able to support authenticating external services (services managed by 3rd parties)
Security	Auditing	At all times, the system should keep tract of any changes to authentication or authorization rules.

As we've seen in the few last scenarios the Identity Provider pattern also helps with security concerns. Table 4.6 below expands on the security aspects of the pattern (using the format that was detailed at the beginning of this chapter).

Table 4.6 Threat categories and actions. List of the action that implementations of the Identity Provider can take and the threats these actions mitigate.

Threat	**Actions**
Spoofing	adding security tokens enables making sure that only authorized requests are handles by a service
Elevation of Privilege	Ensure a service consumer doesn't assert any privileges it doesn't have

The next pattern called Service monitor. Like the Identity provider pattern, the Service Monitor pattern is a mix manageability and security pattern although the security aspect of the Service monitor is secondary.

4.5 Service Monitor

An important aspect of an SOA initiative[1] is governance. If you do not ensure all the different services comply with the guidelines set by the enterprise architect you might not be able to capitalize on the interoperability promise of SOA as well as encounter all sorts of performance, security problems. On top of governess there's the matter of the on going operations of the enterprise. Each service is a small independent system and we need to find a way to manage that and make sure it all works. The Service Monitor pattern helps solve both of these problems. But before I'll go into the details of the solution I want to clarify the problem by introducing two sample scenarios.

4.5.1 The Problem

I already mentioned Governance is very important for an Enterprise. To demonstrate this let me tell you about this one time when a very large organization invited me and a fellow architect, to save their skin. In this organization, let's call it LargeCorp (the real name is withheld to protect the guilty) they deployed a new version of a very important, mission critical, 24x7 system. Shortly thereafter the users of the system started complaining about the poor performance of the system to the point that LargeCorp management stopped most of the development and assigned all it's top developers to try and solve the system's problem. When we arrived we found quite a mess, not only in regard to performance but also in issues pertaining to security, reliability etc. for instance we found that there were a lot of servers whose network cards were set only to 10Mbit instead of 100Mbit, we found that sensitive information was being copies to end-users machines and only then did the system checked if the user was authorized for the information and many other similar problems; They did not make proper use of their staging environment etc.. The amazing thing was that the organization didn't lack the guidelines and procedures that would have prevented this fiasco. They "just" didn't had the means to make sure the procedures were followed.

When we think about a typical SOA initiative it is paramount that we pay attention to governance. Each service is a (relatively) independent and autonomous entity which may utilize a lot of resources like databases, servers etc. If we can't achieve some control over that at the enterprise level we may very well end like LargeCorp mentioned above.

Another even more important aspect of governance and management of an SOA initiative is the on-going operations. Once a system is deployed we need away to make sure quality of service commitments are met, identify security problems, verify liveliness of services etc. Figure 4.X below shows a "bunch of services" that are likely to be found in a typical E-Commerce system. The system has an ordering service, which handles the shopping card until an order is finalized. It then interacts with an Invoicing Service which processes credit cards and other payment methods. The Ordering service also interacts with the warehouse service to secure items or order them from suppliers, a shipping service that monitors the

[1] As explained in Chapter 1 – SOA initiative is an enterprise wide effort to deploy/use SOA

activity until a package is ready – which also interacts with the warehouse and a tracking service with verify an order is fulfilled against multiple shipping companies.

Figure 4.14 typical services in an E-Commerce system. The ordering service interacts with an invoicing service to send invoices and bill customers, a warehouse service to verify an item is in stock or order a new item from a supplier. The Ordering service also triggers a shipping service that, in turn, interacts both with the warehouse service to update quantities and a tracking service that interacts tracking systems of shipping companies to verify that a parcel reaches its destination.

Looking at the relations depicted in figure 4.14 above we can see that the tracing service is not essential to enable to complete an ordering cycle, however if one of the other four services is fails or mal-functions we won't be able to fulfill orders in this system. If we had some way to know when a service is in trouble we can attend it and make sure the business is back on track.

We should remember that the scenario illustrated above is partial and simplified version on what we would usually find on any decently sized enterprise. In this scenario we have 5 services, if each of them has 99.9 % reliability the overall reliability is 99.5%. The reliability decreases the more components we have and if we have 50 services with the same 99.9% reliability our overall reliability will be deteriorate to 95.1 %s (more than 400 hours of unavailability a year). We must have a way to identify problems and fix them fast – we need a way to take a bunch of scattered services and make sure we can maintain an operating enterprise or in other words:

How can we identify problems and faults in services, attend them to ensure the overall business's availability?

One thing to do is to increase the reliability and availability of each service. This can be done by applying patterns like Service Instance or Virtual Endpoint mentioned in chapter 3.

However, while using these patterns will help make each service there's still a chance that something will go wrong and then what do we do. An even more important problem is that a service is rarely truly isolated. Services usually need to interact with other services. So the reliability of each service is also affected by the reliability of the services it has to interact with.

To try to solve this dependency problem we can try to increase the service's autonomy - see for example the Active Service pattern in chapter 2. Nevertheless, the service needs to know if the services it depends upon are down. While an autonomous service can still operate for a while it will eventually be updated.

The next level is to augment the services with internal monitoring and possible adding self-healing capabilities (see the Service Watchdog pattern in chapter 3). However this still leaves a few problems unresolved such as making sure several services follow the same guidelines, identifying problems in the service interconnecting infrastructure (the network that lets the services communicate), system-wide security problems not to mention that one service cannot control and fix problems in other services – especially considering some of them may be external to the organization.

4.5.2 The Solution

There's a limit to what can we achieve and do in the scope of each and every service and like other areas of enterprise management, there's no escaping centralized management or in other words:

> **Apply the Service Monitor pattern and deploy a centralized management point that will monitor services' security, network, QoS, policy and any other governance related issues.**

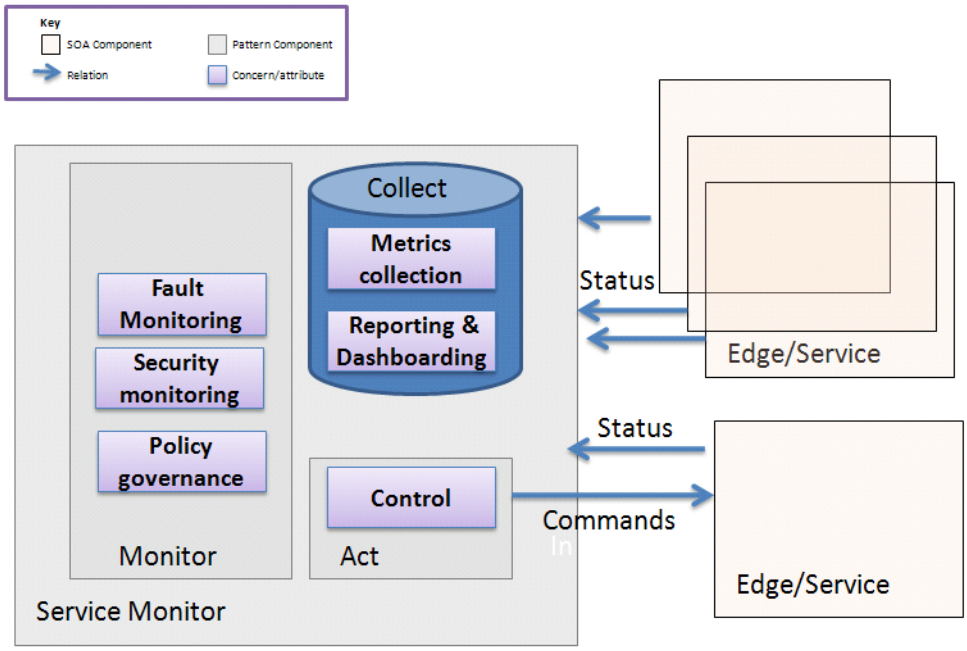


Figure 4.15 The Service Monitor Pattern. A Centralized component – the Service monitor, receives statuses from all the services in the system. The Service Monitor uses that information to infer policy violations, security, performance or other failures and to allow system operators to deal with any of the problems.

As illustrated in diagram 4.15 above the Service Monitor pattern is composed of 3 main components. The basis for everything is the collection component, whose role is to collect and store incoming statuses as well as provide reports and summaries. The Service monitor can gather many types of statuses including for example, performance, faults, number of calls, data transferred etc. As the Service Monitor is gathering this data it can execute different rules to validate and monitor the behavior of the services and make sure they are in order. For instance, the monitor can check the performance figures vs. the promised quality of Service. It can make sure that security policies like “channel encryption” are met etc. Once a problem has been identified the third component of the Service Monitor goes into action and notifies the operators. It can also, send commands to the monitored services, either automatically or through the actions of operators. For example, an operator may choose to restart a faulty service, change the policy for a running service etc.

The Service Monitor pattern is not a replacement for the service self management, increased autonomy options mentioned above it but it helps solve the problems these

options cannot solve by handling cross-services problems like cyclic dependencies, cross-enterprise policies, the entire service dies etc. which can only be identified by looking at the complete pictures. The downside of applying the Service Monitor pattern is added complexity but the gain in systems reliability and manageability make it, in my opinion, essential to all but the simplest systems.

To help the Service Monitor pattern get an overall picture of the services in the system consider combining it with a service registry where The Service Monitor will be able to find the information about the services.

When combined with the Orchestration pattern (see chapter 7) you may also want to enhance the Service monitor with Business process view of the system. Then you would be able to gain the monitoring benefits such as setting and enforcing policies at the process level.

The Service Monitor, which is already a central hub for service interactions can also serve as a central logger (see the Logger pattern earlier this chapter) and help with auditing and debugging.

Service Monitor is not a new concept in the sense that we already have solutions based on similar concepts for non-SOA systems with the most popular ones being CA-Infrastructure Management, IBM Tivoli suite and open source packages like Nagios. However the SOA specific tools adds a few SOA specific features that traditional monitoring tools lack such as handling service's policies.

4.5.3 Technology Mapping

The technology mapping section, as mentioned in the patterns structure in Chapter 1, takes a brief look at what does it mean to implement the pattern using existing technologies as well as mention places technologies implement the pattern. When it comes to the Service Monitor pattern I will mostly discuss the latter and not the first. The reason for that is that implementing a Service Monitor is a relatively big task and, in my opinion it isn't cost effective to pursue it unless you are implementing an internal product that while adhering to most SOA principles uses some non-standard communication technologies.

If you are using web-services and still want to implement the Service Monitor pattern by yourself take a look at the WS-Distributed Management protocol. As mentioned in the technology mapping section of the Service Watchdog Pattern (see chapter 3) this protocol has two sub-protocols. One is management using web-services which is a general management protocol, like SNMP for example, which builds on web-services technologies. The second protocol, known as WS-DM MOWS[2], is more related to the Service Monitor Pattern and it has to do with management of Web-Services

WSDM-MOWS defined contracts and expected behavior supporting services are expected to implement. The Service Manager can then use these to get the status of the service, control the service (e.g. start/stop) and also to subscribe to notification thus allow for

[2] WS-DM MOWS – short for Web-Service Distributed Management – Management Of Web Services

continuous collection of metrics. You can see the further reading section of this chapter for a link to WSDM-MOWS primer by Oasis

Many companies produce SOA monitoring and governance solutions from SOA-specific players like SOA software and general, larger companies like IBM and Oracle. Most of the solutions provides several layers of monitoring starting with the basic network view, which is very similar to what we know from general purpose monitoring solutions. See for example figure 4.16 below which show the network overview of Progress Actional's looking glass SOA monitoring tool. As mentioned above this general view is not too different from what we know from other monitoring suites. SOA monitoring tools provide additional traditional capabilities like auditing and logging.

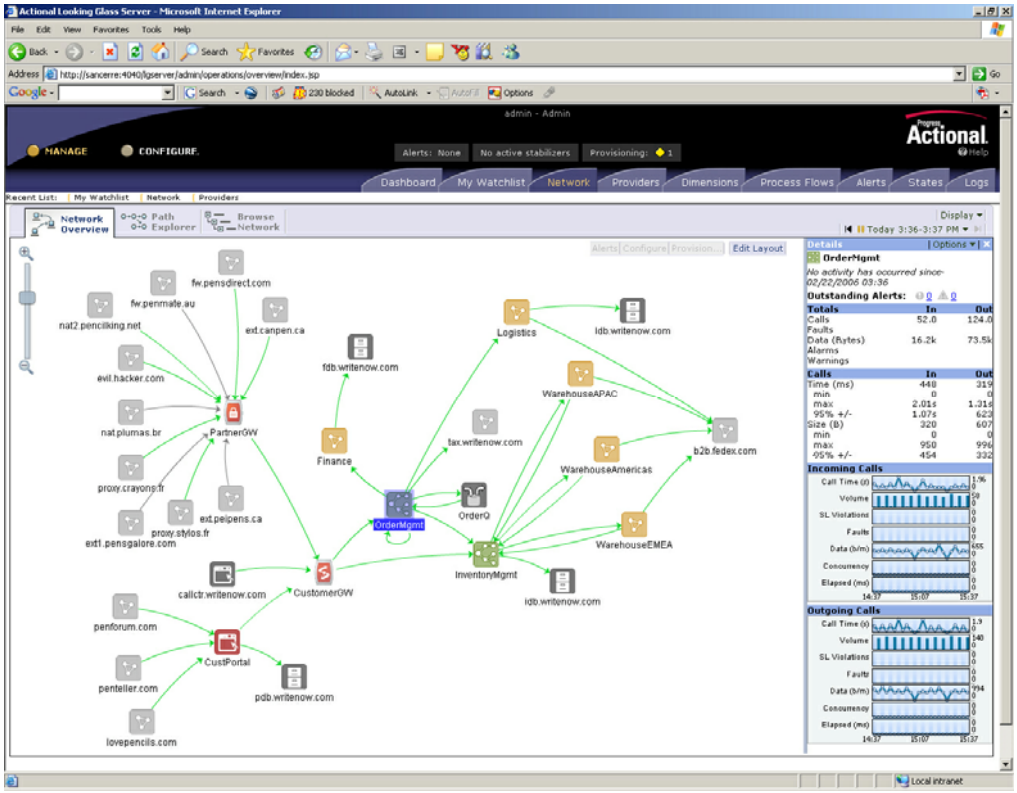


Figure 4.16 the overview panel of one of the dedicated SOA Service management tools (by Actional). In this panel we can see an overview of the service's state, their relations and summary level metrics about the services.

Note that even in the network view we can get some SOA specific information like statuses on calls and performance as well as detect dependencies and cycles. On top of the

basic monitoring features mentioned above SOA monitoring tools add few SOA specific capabilities like monitoring processing time, discovering services, setting and enforcing policies etc. Figure 4.17 below shows the monitoring screenshot from Oracle's Amberpoint. We can see that both the throughput and faults of the Demo manufacturing service as well as the option to examine the WSDL (contract) of the service.

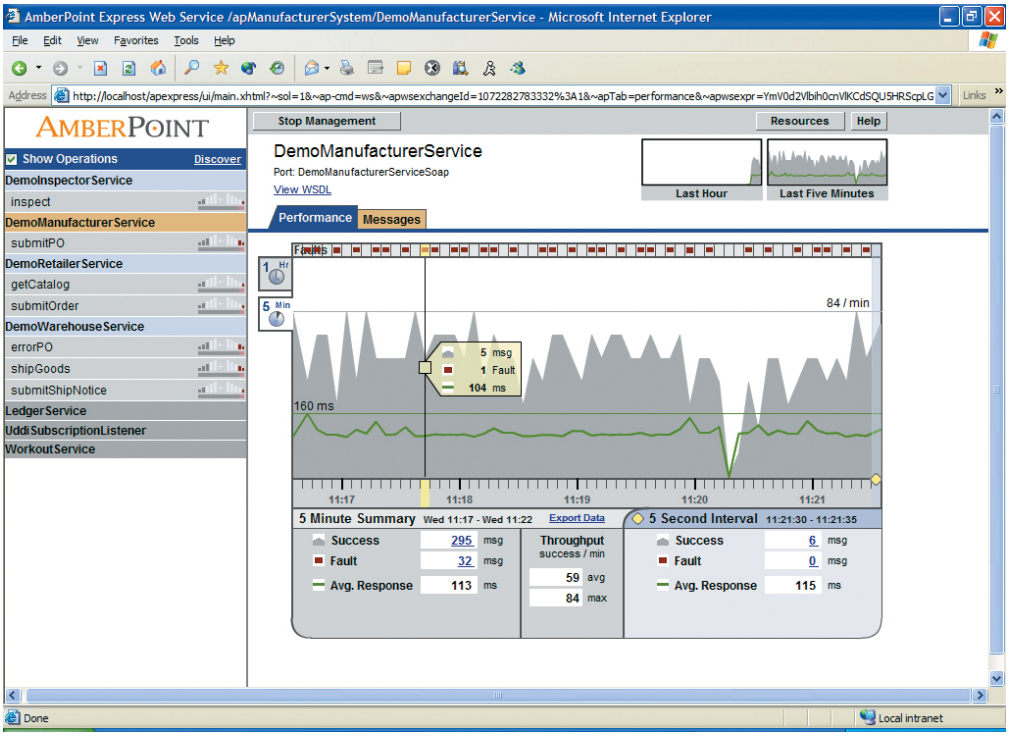


Figure 4.17 – Real-time statistics of a service using the Developer edition of Amberpoint. In this screen shot we can see detailed performance counters, including trends for a specific service.

We've just seen how current technologies utilize the Service Monitor Pattern and let us increase the manageability of our services. I also mentioned that the Service Monitor Pattern can help with security. Let's see how it all connects.

4.5.4 Quality Attributes

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. As was mentioned in chapter 1, most of the architectural requirement are described from the perspective of quality attributes (scalability, flexibility, performance et.) – through the use of scenarios where these attributes are

manifested. The scenarios can also be used as reference for situations where the pattern is applicable.

The main reason to employ the Service Monitor pattern is to get central management and to help make a bunch of services into a working enterprise but it isn't the only reason. The Service Monitor pattern can also help us with testing services before we deploy them, make sure the quality of service is kept once they are deployed, ensuring compatibility between services by making sure their policies match as well as identifying security problem like man-in-the middle attacks.

Table 4.7 below shows a few scenarios where it is beneficial to use Service Monitor pattern.

Table 4.7 Service Monitor pattern quality attributes scenarios. The architectural scenarios that can make us think about using the Edge Component pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Reliability	Mean Time To Repair (MTTR)	Under normal operations the time to discover a faulty service will be shorter than 2 minutes
Manageability	Reporting	At all times, managers will be able to gain an overall view of the status and problems in handling business requests
Testability	Performance	During stress tests we need to be able to time the performance of each service in the system
Security	Governance	During development and operations, the enterprise architecture team will be able to ensure all services use secured channels
Security	Auditing	At all times, the system should keep an audit trail for requestors and their requests.

As we've seen in the last scenario the Service Monitor pattern also helps with security concerns. Table 4.8 below expands on the security aspects of the pattern (using the format that was detailed at the beginning of this chapter).

Table 4.8 Threat categories and actions. List of the action that implementations of the Service monitor can take and the threats these actions mitigate.

Threat	Actions
Tampering	Verifying that all services utilize signatures for their messages
Information Disclosure	Scanning outgoing messages for sensitive content
	Identify man-in-the-middle attacks by watching incoming and outgoing traffic vs. configured routes
Denial of Service	The Service Monitor can compare both performance and number of requests versus the regular or average loads each service has to identify Denial of service attacks.
Elevation of Privilege	Ensure different security policy for internal services vs. external ones

The Service Monitor pattern is the last pattern of the Security and Manageability chapter and appropriately it handles both issues. Let's take a final look at all the patterns covered in this chapter.

4.6 Summary

Chapter 4 took us through several aspects needed to secure SOA implementations. Two of the patterns also have management and maintainability aspects even though they also relate to security. The patterns covered in this chapter include:

- Secured Message – encrypt/decrypt and sign individual messages or message fragments to secure messages when you interact with two or more parties in the same conversation as well as when you only need to secure message fragments
- Secured Infrastructure – Use or create a communication infrastructure that is shared by the service in the organization to provide a secure communications channel.
- Service Firewall – inspect all incoming and outgoing message using software or appliance and help protect your services from several classes of attacks
- Identity Provider – use centralized provisioning and certificate based authentication and authorization to efficiently manage identity in a federated environment
- Service Monitor - monitor and manage services from a centralized location to gain timely access to status of your enterprise.

While these patterns are, in my opinion, very useful and valuable for securing and making your SOA more maintainable, you should keep in mind that making a solution in general and an SOA in particular secure and maintainable goes well beyond these patterns. The patterns listed here deal mainly with the interfaces of your SOA, you still need to make sure the business logic you write is both secure and maintainable especially if the service is

distributed internally. For instance when you log errors/messages or persist data in the database, you should pay attention not to log sensitive information.

I highly recommended you take the time to explore the sources in the further reading section, which includes books like "SOA in Action" or OWASP site that cover additional aspects of security.

Chapters 2,3 and 4 took a look at patterns related to building the services and their interfaces. The next couple of chapters take a look at the interactions of services with their consumers - be these services or humans

4.7 Further Reading

This section provides links to resources (web and books) for all the technologies discussed in this chapter.

Table 4.9 resources for further reading on topics covered in this chapter.

Topic	Resource name/link	Why
	http://www-128.ibm.com/developerworks/xml/library/x-encrypt/	
	http://xml.apache.org/security/	
	http://www.nue.et-inf.uni-siegen.de/~geuer-pollmann/xml_security.html	
Service Monitor pattern	http://www.oasis-open.org/specs/index.php#wsdm-muwsv1.1	
	http://www.opengroup.org/tech/management/arm/	
	http://www.oasis-open.org/committees/download.php/17001/wsdm-1.0-mows-primer-cd-01.doc	
	http://www.openssl.org/	Open SSL
	http://dev2dev.bea.com/pub/a/2005/11/saml.html	Demystifying SAML
	OAuth2	
All	www.owasp.org	The homepage of the "Open Web Application Security Project". A site that has a loto of information on threats and preventive measures.

ALL	24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them by Michael Howard, David LeBlanc & John Viega	Common security problems
ALL	"SOA Security in Action" by Ramarao Kanneganti and Prasad A. Chodavarapu	Security in the context of SOA

4.8 Bibliography

Writing Secure Code, Second Edition, Michael Howard and David LeBlanc

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet - OWASP cross site request forgery prevention cheat sheet

http://www.pathelland.com/presentations/2004/powerpoint/040317-EastCoastRAF-Helland-ThoughtsOnData-04b.ppt Pat Helland "Thoughts on Messages and Data in SOA"

5

Message Exchange Patterns

The first 3 chapters looked at patterns that help us build services and their interfaces like Edge Component, Service Instance and others. Chapter 4 covered how we protect and monitor our services. Chapter 5 is the first of three chapters, that cover the different aspects of services interactions. After all getting services to interact and enable business processes is the reason we want SOA to begin with. Figure 5.1 below illustrates, the chapter's focus is on the interaction of services with their "customers" – the service consumers. A Service consumer is any component or piece of code that interacts with a service. The patterns in this chapter deal with the the basics, i.e. the message exchange patterns. Chapter 6 looks at service consumers and. chapter 7 takes a look at patterns related to service composition and integration.

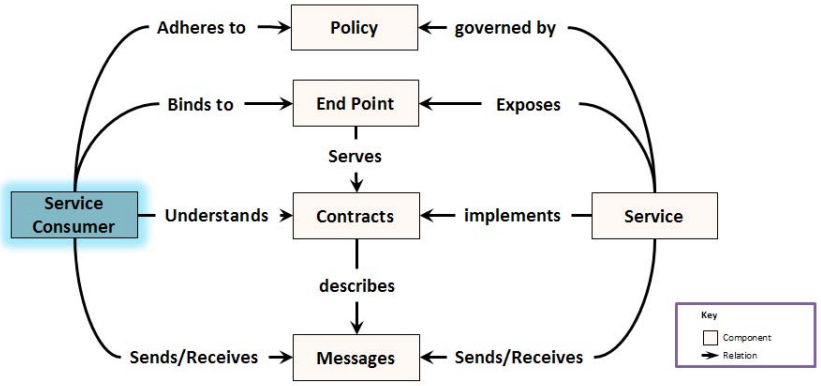


Figure 5.1 Chapter 5 focus is about connecting Services with user interfaces. Chapter 2-4 focused on the different components of SOA, this chapter is the first that takes a look at the service consumers

The SOA definition in chapter 1 says that "Each service expose processes and behavior through contracts, which are composed of messages at discoverable addresses." This makes service interaction very simple, we just, send a message in and get a message back, right? Why do we need a whole chapter or even two on service interactions? Well, it is true that messages are the basic building blocks of service interactions but there are many ways to interact using these building blocks much like the way people use sentences as the building blocks for communications and interactions. For instance, when you call a sales rep you can ask her a specific question and get a reply (request/reply pattern in section 5.2), you can call and leave a question and a telephone number and she will get back to you later (request/reaction pattern in section 5.2); She can call you up and let you know about new products she sells (Inversion of communications pattern in section 5.3); You can have a long correspondence with her company sending emails back and forth until your issues is resolved (Saga pattern 5.4). Indeed, what's true in real life is also true for services.

Note that unlike most of the other patterns in this book, the core interaction patterns existed before SOA was even conceived (see further reading section) – what this chapter will do is look at these interaction patterns from the perspective of SOA and SOA's quality attributes, i.e. what does it take to make an interaction pattern like asynchronous communication and make it work in a way that both complies with the SOA principles and retains the SOA benefits.

Table 5.1 below lists the patterns discussed in this chapter and the challenges that they address

Table 5.1 list of message exchange patterns covered in chapter 5.

Pattern name	**Problem address**
Request/Reply	What's the simplest way for a service consumer to interact with a service?
Request/Reaction	How can we temporally decouple the request from a service consumer and the reply from the service?
Inversion of Communications	How can we handle business events in an SOA?
Saga	How can we reach distributed consensus between services without transactions?

Let's start off with the most basic communications form - synchronous communications. The pattern is called Request/Reply.

5.1 Request/Reply

Request/Reply is probably the oldest pattern in computer science and it has been described many times in the past. For instance, Gregor Hohpe and Bobby Wolfe offer a good description of Request/Reply in their "Enterprise Integration Patterns" book, where they describe the pattern as answering the following question "When an application sends a message, how can it get a response from the receiver". The idea behind Request/Reply in SOA is of course not very different. The reason to discuss this pattern in this book, however, is that there are still a few issues that are worth emphasizing when using request/reply with SOA. I'll talk about these aspects as part of the solution discussion, but first let's take a brief look at the problem

5.1.1 The Problem

When we develop a single-tier software which runs inside a single process on single memory space it is relatively easy to get components to interact. When a requestor component wants something from another component (a replier) it can gain a reference to that replier e.g. by instantiating it. The requestor can then invoke a method on the replier and get the answer- the reply it wanted as a reference or an address in memory where the reply reside. As illustrated in Figure 5.2 below, in SOA, which is an architectural style for distributed systems – suddenly1 the other component is on another memory space and more likely than not on a another machine.

[1] Remote calls have been technologically solved before SOA – but for other architectural styles. Most of these technologies can also be used for SOA – the difference is how you use them. As I said I'll discuss this later in this pattern.

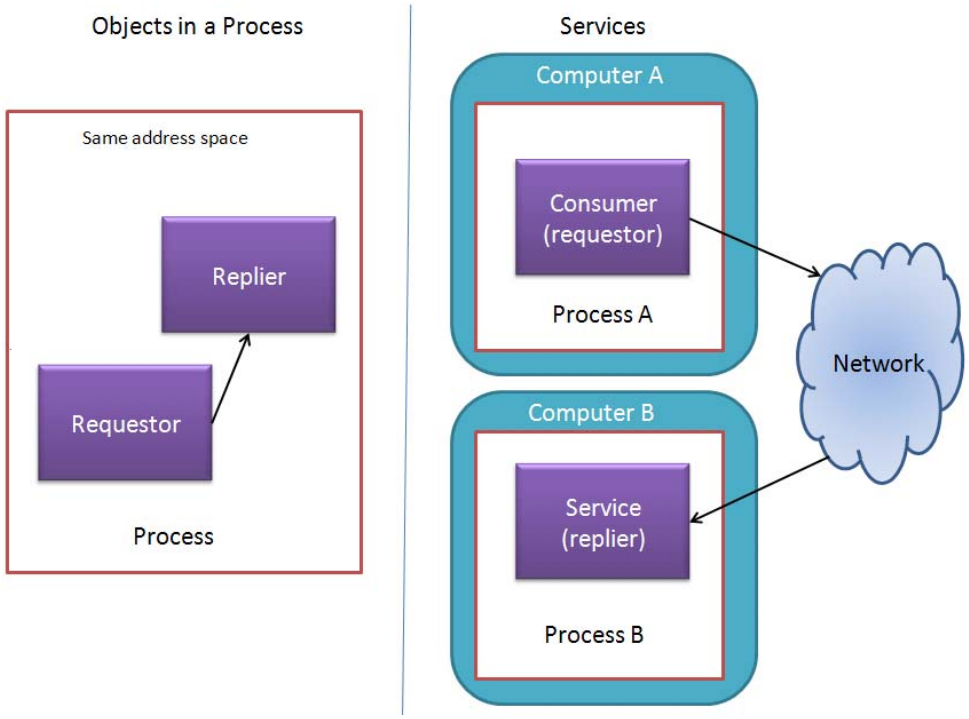


Figure 5.2 local objects vs. Services. With local object making a request from one component to another is simple – you get a reference to the other component and you make a request by calling it. When we deal with SOA the requestor and consumer are not in a single address space. They are also likely not to be on the same computer, maybe not even in the same LAN. Making a request under these conditions is a lot more complicated

So the first thing we want to do is find a way for services to interact with their consumers.

What's the simplest way for a service consumer to interact with a service?

There are several options for service interactions, which are detailed in this chapter asynchronous request/reply long running interactions. They are all more powerful than the Request/Reply. However, there's a price to pay for that - they are more complex than the Request/Reply Pattern.

5.1.2 The Solution

There's a place for sophistication but sometimes you just want to have a simple synchronous interaction between two remote components, to do this we can then

Send a request message from the consumer, handle the request synchronously and send a reply message from the service. Both the request and the reply belong to the receiving service.

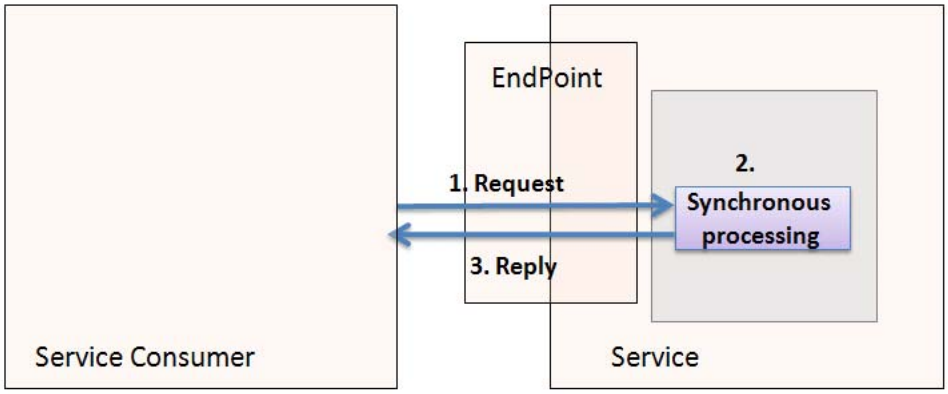


Figure 5.3 The Request/Reply pattern: define both a request and reply messages in the Service's contract. When the service gets a request in the appropriate format, process it synchronously and return the reply message to the service consumer.

The Request/Reply pattern is the most basic interaction pattern so there aren't any special components that make it happen. What you do need is a piece of logic that accepts a request, processes it in a synchronous manner and returns a reply or a result. One thing to pay attention to, is that both the request message and the reply message belong to the contract of the service and not the service consumer (which is a common error I've seen with SOA novices)

Note that service communication infrastructure such as the ones that mentioned in the Service Bus pattern (see chapter 6) handle both exposing the service on a reachable (or even discoverable) address as well as routing replies to senders

Looking at the request and the reply, the roles are again rather obvious. The request holds the intention or the task that the service is expected to perform and the input needed to perform it. The reply holds the results of performing the task

The main problem with the Request/Reply interaction style is that it is suspiciously reminiscent of Remote Procedure Calls (RPC), you know, that DCOM/CORBA , distributed

object stuff. You should be wary of modeling the services' contracts with RPC mindset – this can have several unfortunate affects on your SOA – from poor performance to completely nullifying SOA. Instead of RPC approach you should try to model your contracts with a document centric approach. Sounds good to me – but what in the world is a "document centric approach"? Good question – let's try to understand that together.

In a nutshell, document-centric means that the message contains enough information to represent a complete unit of work and don't instruct the service how to handle the message. On the other hand RPC calls tend to be Command oriented and geared toward sending just the parameters needed to perform the action and have some stateful expectations from the service side as well as implicit expectation on what's going to happen on the consumer side. Document-centric message, on the other hand, don't make these assumptions, having a complete unit-of-work, means that the service has enough information or context in the message to understand all the state it needs. This also means that document centric messages are usually more coarse grained compared with their RPC counterparts. Note that there's a 3rd message type called Event messages – we'll discuss it in the Inversion of Communications pattern in section 5.3)

Table 5.2 below demonstrates 3 ways document centric messages can contain more context.

Table 5.2 options for providing context within a document-centric message.

Context	Explanation
History	The message can contain the interaction up to this point - sort of like breadcrumbs in ht Hansel and Gretel tale. For instance in an Ordering scenario, if the first step was to get customer data and the current step is to set the order (each step can be with another service) the message would contain the customer information when it goes into the ordering service.
Future	The message can include the options the consumer can take to complete the interaction. Again, if we think about an ordering scenario, if the previous step was to reserve the order (see Reservation pattern in chapter 6), the return message can include the information needed to confirm the reservation.
Complete Future	Another way to provide context is for the message format to contain the complete details needed for the interaction. For the ordering example, this would mean that the message would have a skeleton to support all the order and related details and the parties involved will fill in the blanks as the interaction progresses.

Two things to note are that the message can, of course combine more than one type of context and that the same document can be exchanged back and forth between a service

and its consumer(s) – possibly adding detail as it move, to allow complete business processes.

5.1.3 Technology Mapping

The technology mapping for the Request/Reply pattern is rather trivial. Any and all the technologies I can think of enable implementing the request/reply pattern in one form or another. It is important to note, however, that most technologies make it extremely easy to expose objects remotely which encourages RPC style-interactions. and make it hard to get to document-centric interaction. For example the code in listing 5.1 below is an excerpt from the New project wizard for WCF service library in Microsoft's Visual Studio 2010. The sample code shows a developer how to expose to take a simple class and expose its methods as web-services. On the surface it seems like a good example for Request/Reply pattern (except maybe the naming). A service consumer can send the MyOperation1 message with string in it and get the "Hello" concatenated to the string as a reply. However the MyOperation1, implementation is a classic RPC interaction. The situation is a little better for the for the second method (MyOperation2). Here we see a simple document passed to the method. However, the sample code handles that document in an RPC way and doesn't return a document as a reply as well.

Listing 5.1: Code from the new project wizard for createing a WCF service.

```
namespace WCFServiceLibrary1
{
    [ServiceContract()]
    public interface IService1
    {
        [OperationContract]                                              #1
        string MyOperation1(string myValue);
        [OperationContract]
        string MyOperation2(DataContract1 dataContractValue);
    }

    public class service1 : IService1
    {
        public string MyOperation1(string myValue)
        {
            return "Hello: " + myValue;
        }
        public string MyOperation2(DataContract1 dataContractValue)#2
        {
            return "Hello: " + dataContractValue.FirstName;       #3
        }
    }

    [DataContract]                                                       #4
    public class DataContract1
    {
        string firstName;
        string lastName;
```

```
        [DataMember]
        public string FirstName
        {
            get { return firstName; }
            set { firstName = value; }
        }
        [DataMember]
        public string LastName
        {
            get { return lastName; }
            set { lastName = value; }
        }
    }
```

(Annotation) <#1 Exposing a method as a web-service >
(Annotation) <#2 Accepting a document (data contract) as parameter>
(Annotation) <#3 …but handling it in an RPC way and not returning a document as a reply>
(Annotation) <#4 A basic document definition (missing links to related data etc)>

This approach is not unique to .NET as another example we can consider the REST style, while the REST principles promotes the document-centric approach, the basic HTTP verbs are PUT, GET, POST and DELETE which, again make novices think about CRUD interfaces.

A document oriented approach results in richer messages which contain enough context if not the whole of it. Consider for example the XML except in listing 5.2 below. Listing 5.2 shows the result of requesting the a full calendar from Google Calendar. In addition to the calendar details (title, updated, owner name, title etc) you get all the listings with their full detail as well as a pointer to get each calendar entry directly. The result uses Google's GData protocol which in turn builds on the Atom Publishing protocol (APP). Note that the contract for accepting this XML is also simpler than that in listing 5.1 as we need to handle a single XML parameter and not bound the consumers to specific operations that can change over time.

Listing 5.2 A sample document centric reply.

```
<feed xmlns='http://www.w3.org/2005/Atom'
    xmlns:gd='http://schemas.google.com/g/2005'>
<id>http://www.google.com/calendar/feeds/johndoe@gmail.com/private-
0c1e3facdd1a4252aad07effeb7d68cc9/full</id>
  <updated>2007-06-29T19:22:12.000Z</updated>
  <title type='text'>John Doe</title>
  <link rel='http://schemas.google.com/g/2005#feed'
type='application/atom+xml'
    href='http://www.google.com/calendar/feeds/johndoe@gmail.com/private-
0c1e3facdd1a4252aad07effeb7d68cc9/full'></link>
  <link rel='self' type='application/atom+xml'
    href='http://www.google.com/calendar/feeds/johndoe@gmail.com/private-
0c1e3facdd1a4252aad07effeb7d68cc9/full'></link>
  <author>
    <name>John doe</name>
```

```
    <email>johndoe@gmail.com</email>
  </author>
  <generator version='1.0'
uri='http://www.google.com/calendar/'>CL2</generator>
  <gd:where valueString='Neverneverland'></gd:where>
  <entry>
    <id>http://www.google.com/calendar/feeds/johndow@gmail.com/private-
0c1e3facdd1a4252aad07effeb7d68cc9/full/aaBxcnNqbW9tcTJnaTT5cnMybmEwaW04bXMg
bWFyY2guam9AZ21haWwuY29t</id>
    <published>2007-06-30T22:00:00.000Z</published>
    <updated>2007-06-28T015:33:31.000Z</updated>
    <category scheme='http://schemas.google.com/g/2005#kind'
      term='http://schemas.google.com/g/2007#event'></category>
    <title type='text'>Writing SOA Patterns</title>
    <content type='text'>shhh…</content>
    <link rel='alternate' type='text/html'

href='http://www.google.com/calendar/event?eid=aaBxcnNqbW9tcTJnaTT5cnMybmEw
aW04bXMgbWFyY2guam9AZ21haWwuY29t'
      title='alternate'></link>
    <link rel='self' type='application/atom+xml'
      href='http://www.google.com/calendar/feeds/johndoe@gmail.com/private-
0c1e3facdd1a4252aad07effeb7d68cc9/full/aaBxcnNqbW9tcTJnaTT5cnMybmEwaW04bXMg
bWFyY2guam9AZ21haWwuY29t'></link>
    <author>
      <name>John Doe</name>
      <email>johndoe@gmail.com</email>
    </author>
    <gd:transparency

value='http://schemas.google.com/g/2005#event.opaque'></gd:transparency>
    <gd:eventStatus

value='http://schemas.google.com/g/2005#event.confirmed'></gd:eventStatus>
    <gd:comments>
      <gd:feedLink

href='http://www.google.com/calendar/feeds/johndoe@gmail.com/private-
0c1e3facdd1a4252aad07effeb7d68cc9/full/aaBxcnNqbW9tcTJnaTT5cnMybmEwaW04bXMg
bWFyY2guam9AZ21haWwuY29t/comments/'></gd:feedLink>
    </gd:comments>
    <gd:when startTime='2006-08-14T20:30:00.000Z'
      endTime='2012-03-28T22:30:00.000Z'></gd:when>
    <gd:where></gd:where>
  </entry>
</feed>
```

To sum this section, the Request/Reply pattern is supported by all the technologies that allow remote communications. The choice between RPC and Document-Centric is a design decision that is not enforced by the technologies and had to be done by the developers/architects of the solution

5.1.4 Quality Attributes

The Request/Reply pattern is a simple pattern to connect a service consumer with the service that it wants to interact with. As a basic pattern, it doesn't solve a lot of quality attribute concerns except providing the functionality needed (i.e. getting the consumer and the service to interact). One quality attribute that can be important though is simplicity, as a simple pattern it is easy to implement and support and thus help reduce the complexity of the solution. Table 5.3 below shows sample scenarios that can make us think about using Request/Reply

Table 5.3 Request/Reply pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using the Decoupled Invocation pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Time-to-market	Development ease	During development exposing a new capability (already developed) in a service will take less than half a day to implement and test.
Testability	Coverage	During development each capability by of a service should have 100% coverage

I already mentioned Request/Reply is the basic synchronous communications pattern, the next interaction pattern takes a look at implementing asynchronous communications under the SOA constraints and principles. (ck: so we are going from simple to advanced, right?)

5.2 Request/Reaction

Synchronous communication, as described in the Request/Reply pattern above, is very important but it is not enough. The synchronous nature means that the service consumer needs to sit there and wait for the service to finish the processing of the request before the consumer can continue with whatever it is it was doing. There are situations where the service consumer don't want or can't afford to wait, yet the consumer is still interested in getting a reply when it will be available. Clear as mud? Let's take a look at a concrete scenario so I can better explain what I am talking about here.

5.2.1 The Problem

In contemporary border control systems – when travelers get to the immigration officer, the officer then enters the traveler's details and then looks at the passport and tries to match the face to the passport holder. In the last few years countries around the world have begun

the move into e-passport systems. E-passports contains several elements including an RFID chip, machine readable code and a couple of biometric samples (usually face and fingerprints). Figure 5.4 below shows a high-level view of the flow for issuing an e-passport

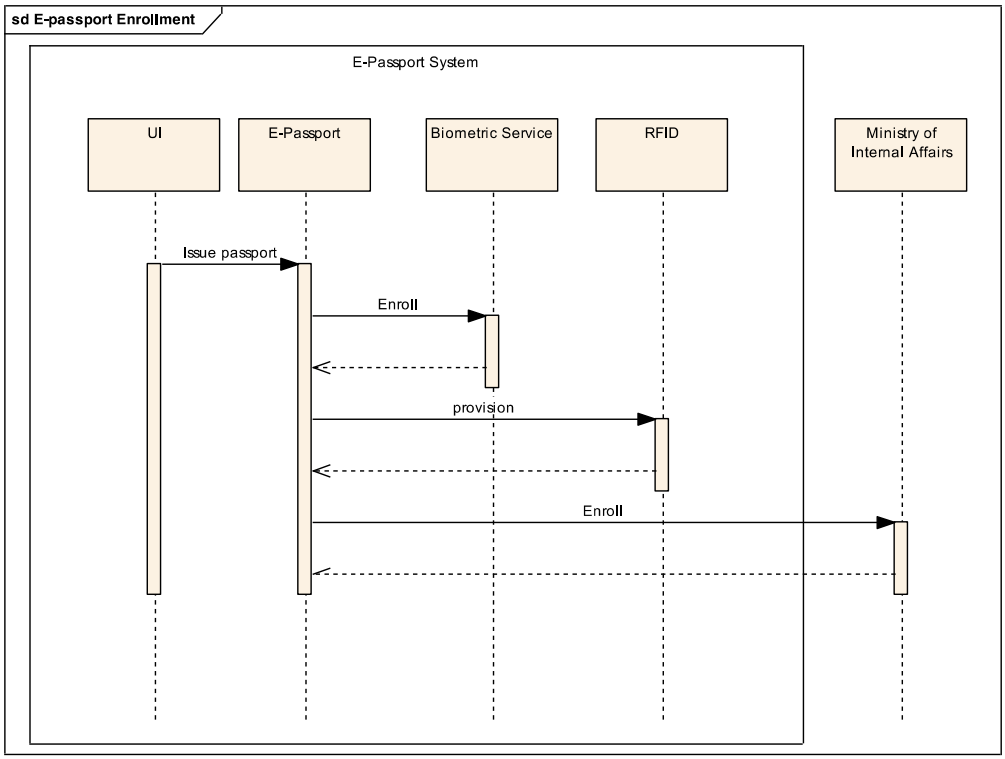


Figure 5.4 a nominal view of enrollment process. When the UI asks the e-passport service to issue a passport, the service has to interact with several other services to fulfill the request. One of the steps in enrolling a person is to check for duplicates. The check can be lengthy but it is essential to complete the process, because of the sequential order of request/reply the other requests have to wait for it to finish.

As can be seen in the figure above, one of the steps in the flow is to enroll the person in the biometric database. While this is not apparent just looking at the interaction the enrollment task can take quite some time to complete. The reason for that is that enrollment also involves checking for duplicates, which is essential to ensure the integrity of the database and prevent mistakes as well as intentional impersonations. The reason it takes time is that you have to compare each sample, e.g. the face against each and every sample already in the database, which can have hundreds of millions of records (the population of the country).

Making this type of request using the Request/reply interaction pattern is problematic because the wait time between the request and the answer is too large – not to mention the time it would take if we decide to do the duplicate checks in a nightly batch.

The situation is not unique to this e-passport system. Similar situations occur in other systems as well, for instance when you buy shares in a trust fund the transaction doesn't happen immediately but you probably want to know when it is completed. Another example is requesting a travel planning system to locate the best deal for your next vacation etc. In order to enable more responsive yet consistent consumers we want to know

How can we temporally decouple the request from a service consumer and the reply from the service?

One option is to solve the temporal coupling on the client side – To do this you spawn a new thread before you send a request to the service – you then let that thread wait for the reply while the consumer goes on with its business. For instance, .NET has a component called BackgroundWorker that performs this separation and allows the UI to dispatch long-running work without blocking the UI thread solution has several drawbacks. For one, the "waiting" is not resilient, i.e. if the service consumer happens to crash, again the reply would be lost when the consumer wakes up again. Additionally the thread takes up resources on the consumer – what if the request takes hours or days? Lastly it is a matter of responsibility. The service is the one which has a task that is time consuming – it should be its responsibility to solve the matter and not throw it at its consumers.

Another approach to solve the temporal decoupling is to circumvent it and break the interaction – for example, usually when we order an item on-line we don't sit and wait until the system ships the item to us – rather the system lets us know the item was ordered. Registering the order takes much less time than fulfilling the order. The downside here is that we don't know if it has shipped or not – unless we try from time to time to check the order status – again like in the previous approach it is our responsibility as service consumers to solve the shortcomings of the service.

There are interaction solutions that support complex interactions, like the Saga pattern, which appears later in this chapter (section 5.4) – while implementing a Saga will be able to solve this issue it is like killing a fly with a cannon - overkill, when all you really need is a delayed replay.

5.2.2 The Solution

When Saga is overkill, breaking the integration works but hurts the service consumers and we want to avoid client side integration for its bad implications. What we really want to do is to somehow implement asynchronous communications over SOA and do that in the simplest manner. Indeed, we can do that, what we need to do is :

Introduce the Request/Reaction pattern and implement asynchronous communication between service consumers and the service. Implement the message exchange as two one-way messages - one (request) from the consumer and one (reply) from the service side

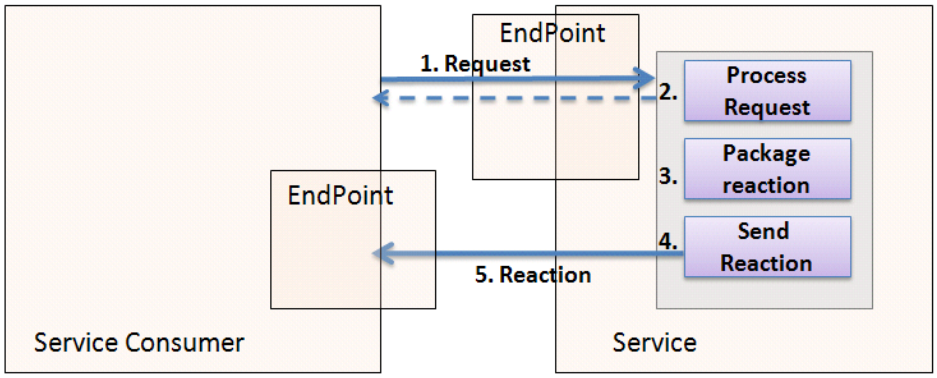


Figure 5.5 The Request/Reaction pattern: define both a request and reply messages in the Service's contract. When the service gets a request it processes it and prepares a reaction. When the reaction is ready the Service send the request back to the consumer

The idea behind the Request/Reaction interaction pattern is to have two distinct interactions between the Service Consumer and the Service. The first interaction sends the request to the server, which may return an acknowledgment, a ticket or an estimate for finishing the job to the consumer. Once the processing is complete the service has to initiate an interaction with the service consumer and send it the reply or reaction (note that the service has to manage the knowledge on where to return the reply – we'll discuss later on).

The Request/Reaction pattern is more aligned with the basic premise of messaging (vs. Request/Reply which is more aligned with RPC) as it lifts the time coupling.

Figure 5.6 below shows the interaction with the biometric service, when employing the Request/Reaction pattern. Now when the biometric service receives an enrollment message it reacts with an "enrolling" message. The "enrolling" message notifies the client that the request has been received. Once the service finishes the enrollment either successfully or with an error) the it would prepare an enrollment reply with the enrollment records and would send it to the client

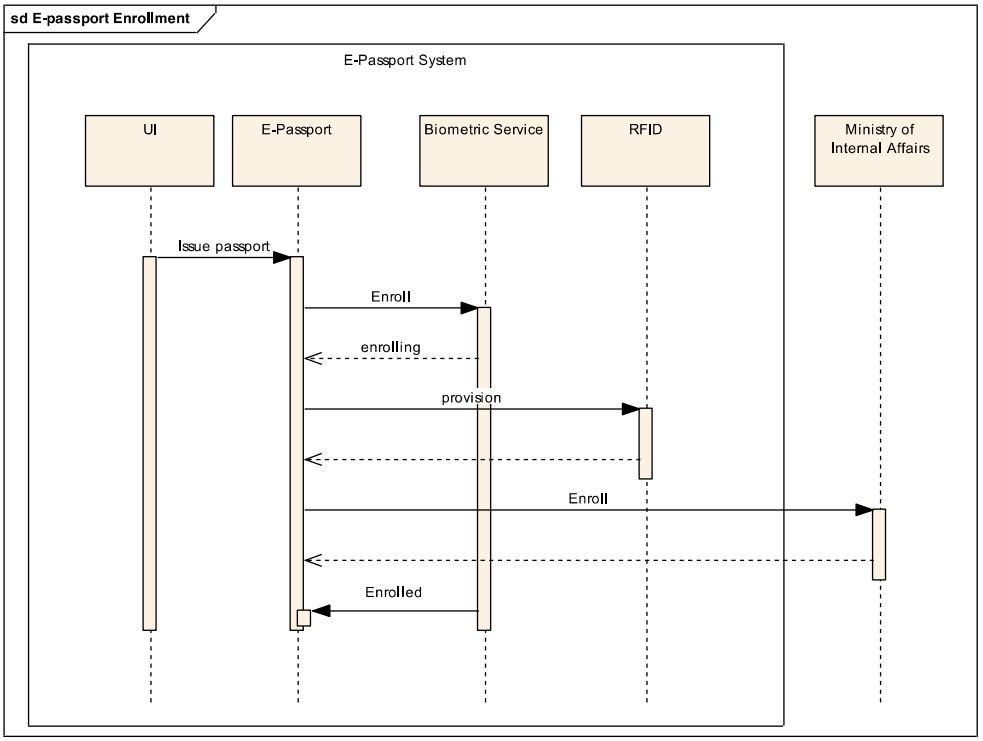


Figure 5.6 The passport issuing process using the Request/Reaction pattern. Now the biometric service returns 2 messages. First it returns an acknowledgment that it processes the message and then when the process is finalized it returns a status. Note that in this cases you may have to use patterns like the Saga pattern (see 5.4 below) to rollback/compensate with the other services if the duplication check finds a duplicate identity.

The request/reaction pattern is used, for example, in the Decoupled invocation pattern (see chapter 2), the difference between the two patterns is that Request/Reaction decouples the response from the request and the decoupled invocation also decouples the processing of the message.

The interactions semantics of the Request/Reaction pattern are limited. For instance if the scenario above also included the possibility to cancel the enrollment if, for example, the RFID provisioning failed it would be problematic coordinating this with a bunch of request/reactions. In these long running interactions you may want to consider more advance patterns such as the Saga pattern described later in this chapter (section 5.4)

The Request/Reaction pattern offers more flexibility than the Request/Reply pattern. However, this flexibility comes with a price – the Request/Reaction pattern is more

complicated then Request/Reply and requires more work on the Service (or Edge) side. Let's take a look at some of the implementation details that we need to take care of.

5.2.3 Technology Mapping

The basic way for implementing the Request/Reaction interaction pattern is to use two one-way messages. If you are using web-services it would mean two http channels, if you are using messages then you'd need a queue (endpoint) for each of the involved parties.

The first hurdle to cross comes from the temporal decoupling. As the request and the reaction (reply) are separated in time other messages can get in between. This means that you need to provide a way for the service to know where to send the reaction. It also means that both the service and the service consumer need to have a way to correlate between the request and the reaction messages - see the correlated messages callout for more details.

CORRELATED MESSAGES

One challenge of asynchronous messaging comes from the fact that the reply message (i.e. the reaction) and the request are not directly related. The reaction can arrive quite some time after the original request was sent. What we need in this case is a way to identify that the two messages are related. The mechanism to solve this problem is known as correlation identifier and as the name implies it has to do with adding a token to all the messages that service consumers and services can use to identify the related messages. This is not very far from the idea of session cookies in a web-application. The correlation identifier can include a message id, a token for the conversation etc.

Correlation is supported by a wide variety of the WS-* standards, for instance WS-Addressing has a relationship message id header which can be used for correlation. Another example is WS-BPEL which has even better support for correlation by letting developers define multiple correlation sets as well as letting developers control the content of these sets.

Both Java and .NET offer solutions to deal with one-way messages. The apache Axis2 Java library, even provides the infrastructure to implement the complete Request/Reaction pattern out of the box. You can see the code excerpt in listing 5.3 below for the consumer side code needed to send an asynchronous message.

Listing 5.3 sample client code for sending a message to a service using the Request/Reaction pattern using on-way massages

```
boolean useTwoChannels = true;
.
.
.
OMElement messageBody = helper.FormatmMessage(data,type);
```

```
Call msgSender = new Call();
msgSender.setTo(
        new EndpointReference(AddressingConstants.WSA_TO,
                "HTTP://www.example.org/ServiceName));
msgSender.setTransportInfo(Constants.TRANSPORT_HTTP,
    Constants.TRANSPORT_HTTP, useTwoChannels);
Callback callback = new Callback() {
        public void onComplete(AsyncResult result) {
         //code to handle the Reaction goes here
        }

        public void reportError(Exception e) {
         //code to handle errors..
        }
};
msgSender.engageModule(new Qname("addressing"));
msgSender.invokeNonBlocking("MessageName", messageBody, callback);
```

While from the architectural point of view, the reaction is a message that is sent by the service. From the implementation point of view, it can also be implemented using pulling from the Service Consumer. Implementing Request/Reaction on top of Request/Reply is not too complicated. Figure 5.7 lists the steps. When the Service Consumer sends a request it will get as a reply the address (URI in this case) of the reaction. The consumer will also get a time token designating the time when the answer will be expected. Once that time elapsed (you can use the Active Service pattern to keep track of time– see chapter 2) the consumer will make a second request to the Service – this time asking for the reply – e.g. using the GET command.

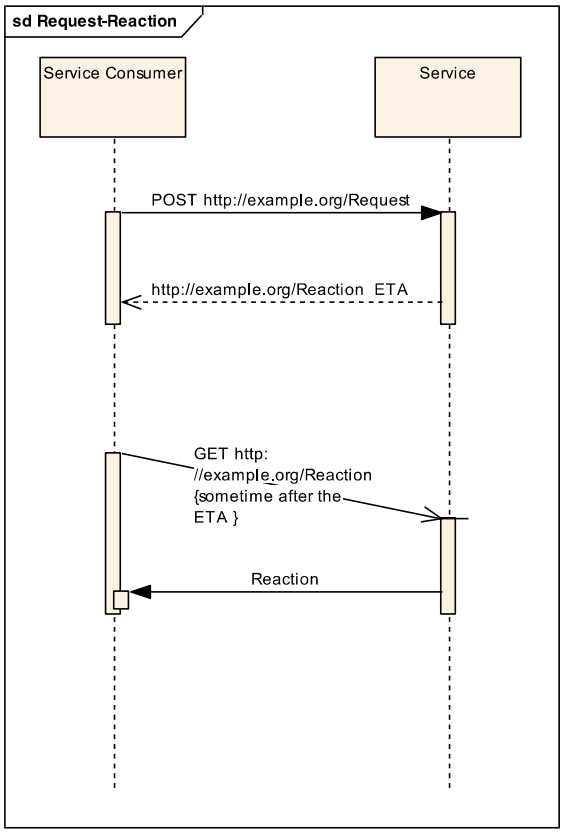


Figure 5.7 Implementing Request/Reaction on top of Request/Reply. The Requests return message explains where to find the reaction the estimated time or arrival (ETA). Sometime after the ETA, when the Service Consumer isn't busy, it can go to the Reaction address on the Service and obtain the reaction itself.

The reason to go down this path (of using pull instead of push) is when you cannot create an active independent endpoint on the consumer side. Again, the preferred way is to get Request/Reaction pattern right – However if you can't you can implement this approach and still conform to the general idea behind the pattern – which is to offer flexibility and temporal decoupling.

5.2.4 Quality Attributes

I've already mentioned temporal decoupling and the flexibility it brings are the main quality attributes that drive using the Request/Reaction pattern. However the pattern can also help with the performance quality attribute. When the sending a message to the Service

doesn't block the service consumer it allows it to allot CPU cycles to work on its own problems (e.g. handling requests from other services). Compare that with the blocking Request/Reply pattern which holds resources on the consumer side while it waits for the reply.

Table 5.4 below shows a couple of sample scenarios where request/reaction is more applicable than other patterns.

Table 5.4 Request/Reaction quality attributes scenarios. These are the architectural scenarios that can make us think about using the Decoupled Invocation pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Flexibility	Temporal coupling	Under normal conditions, notify the ordering party about order shipment within 2 hours after shipping the package.
Performance	Responsiveness	Under normal conditions, the UI will not hang while long operations are performed (such as searches, course recalculations etc.)

The Request/Reply pattern demonstrated synchronous communications between Service Consumers and Services. The Request/Reaction demonstrated Asynchronous communication, what's left is to check if we can accomplish communications using event driven architecture without violating any SOA constraints and assumptions.

5.3 Inversion of Communications

Request/Reply and Request/Reaction patterns are geared toward interactions where the consumer wants to get some information or action from a service. In order to get the action or information the service consumer is willing to pay the coupling price which is associated with knowing about the other service, the service capabilities and the protocol (contract) it uses to expose these capabilities. What happens when the potential consumers don't know they need to go and ask a service for new information which is now available ? Will the Service let them know? Will it be willing to pay the coupling price mentioned above?

Maybe at first glance you don't think that the situation mentioned above is likely to happen - but let's look at a few examples and we'll see that it is a common enough business situation, maybe even the norm.

5.3.1 The Problem

Let's think of a air traffic control system of an air-line. Let's say that we want to create a service that would pro-actively try to take care of delayed flights. For instance when a flight is expected to arrive late we would want to try to find a new flight for transit passengers who won't make their connections (yeah, right..) and free their places in their current connecting flights and adjust the rates for these flight (which does happen). To do that we have to interact with several services some of them would be part of our system - e.g. a service that tracks all the active flight, and some would be external to our system e.g. weather reports, air port statuses either from each airport or from a central authority. For instance, figure 5.8 shows delay information we can get from the FAA's in the US.

AIRPORT STATUS INFORMATION

provided by the FAA's Air Traffic Control System Command Center

Dallas/Ft Worth International Airport (DFW) Real-time Status
The status information provided on this site indicates general airport conditions; it is not flight-specific. Check with your airline to determine if your flight is affected.
Delays by Destination: No destination-specific delays are being reported.
General Departure Delays: Traffic is experiencing gate hold and taxi delays lasting 15 minutes or less.
General Arrival Delays: Arrival traffic is experiencing airborne delays of 15 minutes or less.

This information was last updated: **Jun 21, 2007 at 1:54 PM GMT+00:00**

Figure 5.8 Arrival and departure delays information as provided by the FAA. This can be a source of information for an airline traffic control system. Source **http://www.fly.faa.gov/flyfaa/usmap.jsp**

Figure 5.9 shows our delays service and few of the services it can consume to be able to work its magic

Figure 5.9 Some of the services that a service which handles Delays needs to interact with. The delays drives some of the services directly (e.g. reservations, schedules) but is driven by data coming from the other services (weather, operational picture and airports).

By the way, if you google for business events, you would notice that airlines examples are quite popular but it is important to note that there are many other, more down to earth (pardon the pun), run-of-the-mill IT examples – just think about stock prices or someone who needs/wants to know every time an order is larger than X amount of dollars; Ordering of new parts when the supply is getting below a certain threshold; Any dashboarding and business activity monitor solution (BAM) etc.

While SOA seems to be rooted in request/reply, I think it is pretty obvious we also need to find a way to somehow support business events within the SOA constraints and tenets or in other words:

How can we handle business events in an SOA?

One option is to stick with the base SOA approach and have the service that generate the event actively send a message to all the interested service. Note that the source service has to know about all the interested services, which includes understanding their contracts to support this scenario. This is problematic as it introduces needless coupling between the event source to other services. We need to remember that unlike a classical Request/Reply scenario, the source service doesn't care about the target services. So that in the example above the weather service will have to know about the delays and operational picture services. Oh yes, and what if the airport status also wants to know about the weather so now, we have to change the weather service to notify the new service as well.

Another option is polling by the interested services. Every event basically has a time span when it is still evident in the current state of the service. An interested service can just poll the event generating service and find out about the interesting events. The advantage over the previous option is that now the dependency direction is correct. The services that do the polling are the ones interested in the information. The problem with polling, however, is that chances are you either have the polling interval too large and you'd miss important events or, alternatively, have it too short and you'd cause unnecessary network loads (we can overcome this caveat – and I'll talk about this as a variation on the solution below).

We can alleviate the service's coupling problem, mentioned in the polling option above, if we externalize the relationship from the services. One way to do this is using the Orchestration pattern (see chapter 7). In a nutshell, using the Orchestrated Choreography means we have an external workflow engine. The event source can then have a single dependency – on an endpoint of that workflow engine. The workflow has the knowledge of all the interested parties and as part of handling the incoming message – forwards it to them. This is a step in the right direction since the services are not coupled and it is easy to make changes to the workflow and add additional services. The downside is that we just federated our logic between the services and the workflow.

We've seen three different solution each has some advantages but maybe we can do better ?–Well, I think we can.

5.3.2 The Solution

The answer to handling business events was actually there in the background all the time. We want to add events? Why not adopt an architectural style that is built around events and incorporate that to SOA? As it happens, we don't have to invent the wheel – there is such an architectural style already and it is called Event Driven Architecture (EDA).

Event is any significant change that happens within the event generator or a component that is observed by the event generator. Event specifications in EDA are structured entities akin to SOA contracts and messages. An event specification is made of a header and a body, where the header contains the meta data and the body the actual information on the event. Unlike traditional messages events do not have a specific destination. EDA is similar to publish/subscribe but it also has several differences like historical perspective by treating events as streams instead of isolated occurrences.

To solve our need for event-like message exchange pattern within SOA we can then

Implement the inversion of communications pattern by supplementing SOA with Event Driven Architecture (EDA) , i.e. Allow services to publish events streams of changes that occur in them instead of calling other services explicitly .

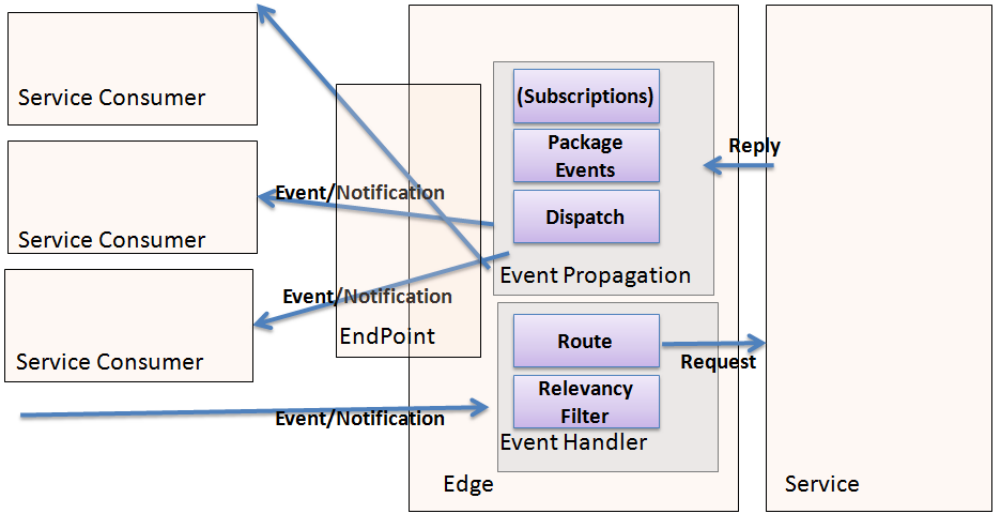


Figure 5.10 The inversion of communications pattern. The Service's Edge accepts and filters incoming events in addition to "standard" requests. When the service has some reply or reaction to an event ready the Edge also packages and dispatches it as an event to service consumers.

The Inversion of communications pattern got its name since it basically reverses the direction of the information flow. Instead of the service consumers calling on the service to get information, the service reaches out to them with new updates. This change in roles requires two components within the service or rather the edge (since they are not really business oriented). The first component is for event propagation. Events should be packaged

in the format agreed upon for the SOA initiative or if there is no common contract according to the service's contract and distributed (see the technology mapping section for some discussion on this). The second component, the event handler, has to do with the service acting as a service consumer for events sent by other services. The first task for the event handler is to filter incoming events for relevancy. This is important since many of the events received might not be relevant, especially if the infrastructure between services is not smart enough to route and/or manage subscriptions. The second role of the event handler is to route the relevant events to the service's components that can react to the events – i.e. the components that under request/reply thinking would get the new information as requests.

One thing you have probably noticed is that even though the Inversion of Communications pattern talks about events it does not include a subscription management component on the Edge – or. The reason for that is that subscriptions management requires too much effort which is not really related to the services like routing, persistent subscriptions etc. An alternative to subscriptions on the service is to move the responsibility to the consumer and/or infrastructure. To do that you can provide "known names" (e.g. URIs, queues etc.) where events can be found and then have interested services listen to them.

Looking back at our Delays service mentioned in the problem description we can see (Figure 5.11 below) that now the Airports, Weather and Operational picture services push their changes to the Delays service instead of the other way around. This has a positive effect on network traffic as the delays service no longer has to worry about missing an important change in the three services it monitors. Also note that applying the Inversion of Communications pattern does not have to mean you move all your interactions to events. In our example the Delays service still makes request/reply interactions with the Scheduling and Reservation services – e.g. identifying a delay it can try to reserve places on later flights for people who would miss their connections.

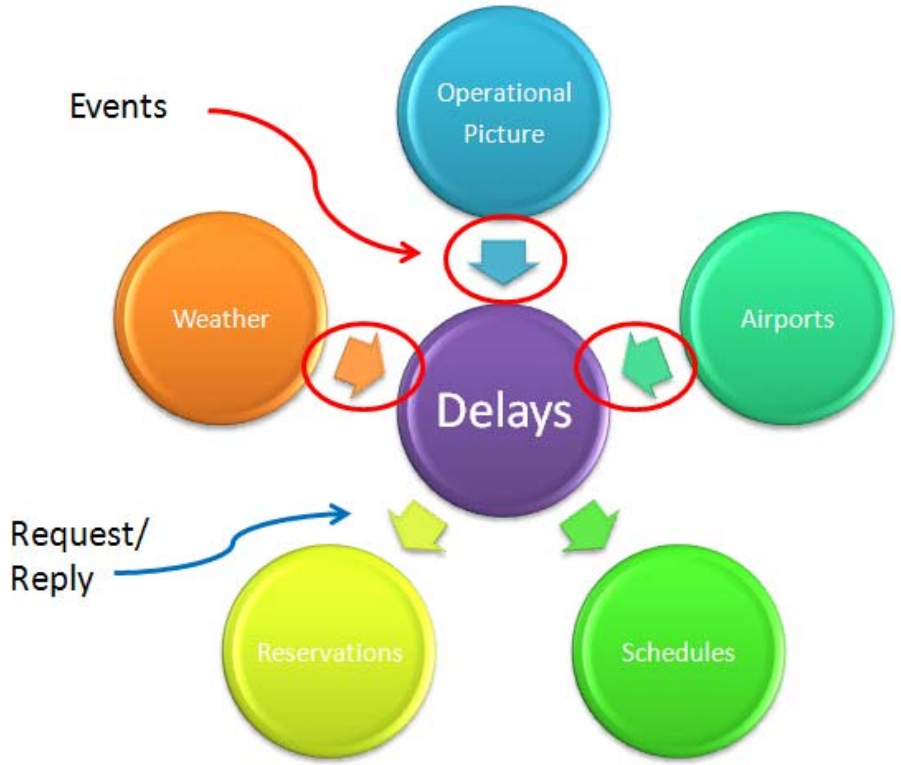


Figure 5.11 The Relations between the Delays service mentioned in Figure 5.9 when moving using the Inversion of Communications pattern. Now the Weather, Airport and Operational Picture services push their changes to the Delays Service.

One thing to note when we combine the Inversion of Communications pattern with the Request/Reaction or Request/Reply patterns is that in addition to replying to the service consumer (or as an alternative to doing this) we should also send an event to propagate the effects of the request handling.

Inversion of communications is about implementing EDA on top of SOA. Up to now we've seen the simple side of that which is handling sporadic or isolated events. However a very strong concept that EDA defines is event streams. Event streams means we don't just look at each event on its own, but rather as a chain of related events spread in time. Event streams can give us both trends and historical perspective. Used right, this can give us real-time Business intelligence and business activity monitoring. You can take a look at the Aggregated Reporting pattern in chapter 7 to see an application of this capability.

Another pattern you can combine the Inversion of Communications with is the Parallel Pipelines pattern (see chapter 3). This combination can be used to provide an SOA implementation of a Staged Event Driven Architecture (SEDA) .In a nutshell SEDA can provide you with a way to increase concurrency and throughput of a solution in a relatively simple way

The downside of using Inversion of Communications is the added complexity of thinking in events on a system-wide scale. The way to elevate this problem was already mentioned – don't use Inversion of Communications exclusively, rather, combine it with the other message exchange patterns mentioned in this chapter.

One other thing to watch for when using the Inversion of Communications pattern is avoiding a vicious event circle, where an event triggered cause a chain of events that gets back to the original event source and cause it to re-fire the same or similar event. I can't say I've seen it happen in real business scenarios but the possibility exists. The way to handle this problem is logging and monitoring (it is hard to try to predict it since you cannot plan who the event consumers will be) – see for example the Service Watchdog pattern in chapter 4.

Moving to Inversion of Communications also makes it more complicated to debug processes. When something goes amiss you need to trace back the butterfly whose wings initiated the chain reaction that led to the problem. The way to counter this is via centralized logging through the development process (and possibly in production) that will enable replay of the system. Nevertheless it is still more complicated than just following a direct call stack.

Another challenge of moving to Inversion of Communication is adding it in the middle of an SOA initiative, i.e. when you already have quite a few services deployed that utilize the simpler message exchange patterns. I can't provide a general guidance on the interaction remodeling since it is very specific – but I'd say that like the SOA initiative itself, the secret here is to perform the transition gradually.

The other set of challenges related to the Inversion of Communications pattern has to do with the implementation details – after all many of the SOA infrastructures (most obviously HTTP) don't support events or multicasts. Let's see if we can clear these obstacles…

5.3.3 Technology Mapping

There are several technology mapping options for implementing the Inversion of Communications. The first option, which is also the most natural fit, is using an ESB. Most ESB implementation provides implementation of all the common message exchange patterns including publish subscribe. For instance listing 5.4 below shows how to configure a subscription on Apache ServiceMix (an open source ESB). To configure the subscription you just add subscriptions section (sm:subscription) in the configuration section of a component (sm:activationSpec)

Listing 5.4 An excerpt from a configuration file of a component in Apache ServiceMix. The configuration includes a subscription for a "picture" component

```
          <sm:activationSpecs>
              <sm:activationSpec componentName="sub"
   service="foo:Subscriber">
                .
                .
                   <sm:subscriptions>
                        <sm:subscriptionSpec service="cop::picutre"/>
                    </sm:subscriptions>
              </sm:activationSpec>
          </sm:activationSpecs>
```

Thus, basically, it when you want to implement the Inversion of Communications Pattern with an ESB you delegate the responsibility of passing the events as well as managing subscriptions to the infrastructure and you can concentrate on planning the events and the other business activities.

Even though it isn't a common service infrastructure for SOA, you can get even more loose coupling by using a messaging infrastructure (or ESB) that supports topics. Topics are more loosely coupled since the subscribers don't even know who the publisher is they just know about the topic that they find interesting. The problem with that off course is that the subscribers don't even know who the publisher is so we need the infrastructure to make sure only authenticate and authorized services can post events.

Ok, but what's with the more problematic infrastructures like HTTP (e.g. RESTful services) or even plain TCP? Well there are two options here. One option is to write the infrastructure needed as part of the Edge component of each service. In other words develop logic to persist subscriptions and actively send each generated event to all the interested subscribers. While it is technically feasible, I don't recommend going down this path unless you are a middleware vendor. For the rest of us it is better to focus on our core business and business value for our solution and not try to develop a delicate piece of infrastructure we are not likely to get right on the first try anyway.

The second option, which I find more interesting, has to do with a push, er . pull, application which you probably already use daily – blogs and blog news readers. When I publish a new event (post) in my blog it isn't immediately sent to my blog subscribers. In fact it is never actively sent. Instead, the new event is added to an events stream (RSS or ATOM feed) which contains the most recent events. The subscribers, who manage the subscription on their side without any regard to me (loose coupling) , decide how often they need to poll my event steam so that they wouldn't miss important events – based on how many items I keep in my feed, the frequency of new events and the latency in handling the events they can afford. Note that consumers that need low latency from event occurrence to

notification would probably need the on-line event notification and won't be able to use this method.

As we've seen in the Request/Reply pattern (section 5.1 above) ATOM Publishing Protocol is already a popular choice for formalizing collection in RESTful web services as are JSON versions of it like OData and GData..

Events Time-To-Live

Whether you use feeds or a queue based approach for publishing events you need to consider the event's time-to-live (TTL). By TTL I mean the time that the event should be available to various consumers to consume before it becomes irrelevant.

When we use events in a programming language the TTL is easy –" you snooze you lose" - if a consumer is not there when the event is raised it is its problem. In an SOA it is more wise to allow temporal decoupling between the time the event was raised and the time it is consumed. This temporal decoupling allows increased autonomy and loose coupling for both the event generator and the event consumer. The flip side is that we now have to consider the TTL of events to prevent processing of obsolete information, too much latency and performance problems.

The TTL changed depending on business meaning of the event so there aren't any firm rules. Two thumb rules I can give are that the TTL for cyclic events like stock price update, is usually the cycle frequency and that the TTL for one-time events like a new order, tends to be much longer.

One point mentioned briefly in the previous section was that the EDA part of the Inversion of Communications pattern allows us to treat events as a stream rather than isolated instances. Event streams can also enhance your solutions even more if you add additional architectural concept known as Complex Event Processing (CEP). As its name implies, CEP means taking a look at event streams and examining it for complex patterns. This is probably best explained through an example. Listing 5.5 below shows a sample query in an embeddable CEP engine I wrote a few years a go (which was based on C# Linq). The query examines a stream of login events and raises an alert whenever there are 3 failed logins in a row from the same user.

Listing 5.5. sample continuous query to look at a stream of login events and raise an event when there were 3 consecutive failed logins.

```
var loginRecords = engine.GetEventSource<Login>();


engine.AddQuery(() => from names in loginRecords.Stream
```

```
                        group names by names.Name
                        into logins
                        from login in logins
                        let next = logins.FirstOrDefault(t => t.LoginTime >
login.LoginTime)

                        let nextNext = null == next ? null :
logins.FirstOrDefault(t => t.LoginTime > next.LoginTime)
                        where
                          !login.Successful && (null != next &&
!next.Successful) && (null != nextNext && !nextNext.Successful)
                        select login, HanleAlert);
```

There are many commercial CEP engines from companies like SAP, Tibco and IBM as well as few open source options like ESper from EsperTech.

Inversion of communications pattern presents a good opportunity to introduce CEP to a project, however it isn't the main reason to use it. Thus, as usual, we'll finish our discussion of the pattern by exploring some of the motivations for using it..

5.3.4 Quality Attributes

The Inversion of Communications is a powerful pattern. Events based interaction greatly help increase the autonomy, composability and reuse within the system. This is a great news to an SOA, so much that Gartner for example, called EDA and SOA "Advanced SOA". While it is important to remember the challenges involved in its implementation like complicated debugging and the added work in designing events (see the pattern description above) it is an important pattern to have in your tool set because all of the benefits. Table 5.5 below shows some of the scenarios to make you think about using the Inversion of Communications pattern.

Table 5.5 Inversion of Communications pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using the Decoupled Invocation pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Flexibility	Decoupling	Services should know as little as possible about each other
Reuse	Interfaces	All services should support <some common service APIs> in addition to any specific requests they may serve.
Changeability	Add feature	Integrate a new capability into the system in 3 calendar weeks or less.

The Inversion of communications wraps up the basic message exchange patterns by showing how to do eventing or publish/subscribe within SOA. The last message exchange pattern we'll cover in this chapter is the Saga pattern, which shows us how to get transactions-like behavior beteween services.

5.4 Saga

Back in chapter 2, we talked about the Transactional Service pattern as a way to help make a service handle requests in a reliable manner. However, using the Transactional Service Pattern, only solves one part of puzzle. Let's take another look at the scenario that we looked at there and see what we still need to solve. Figure 5.12 below, shows an Ordering service that processes an order. The interesting issue here comes from steps 2.3 and 2.4. Within the internal transaction of handling the request, the Ordering service has to interact with two other services: request a bill from an internal billing service order stuff (e.g. parts or materials) from an external supplier.

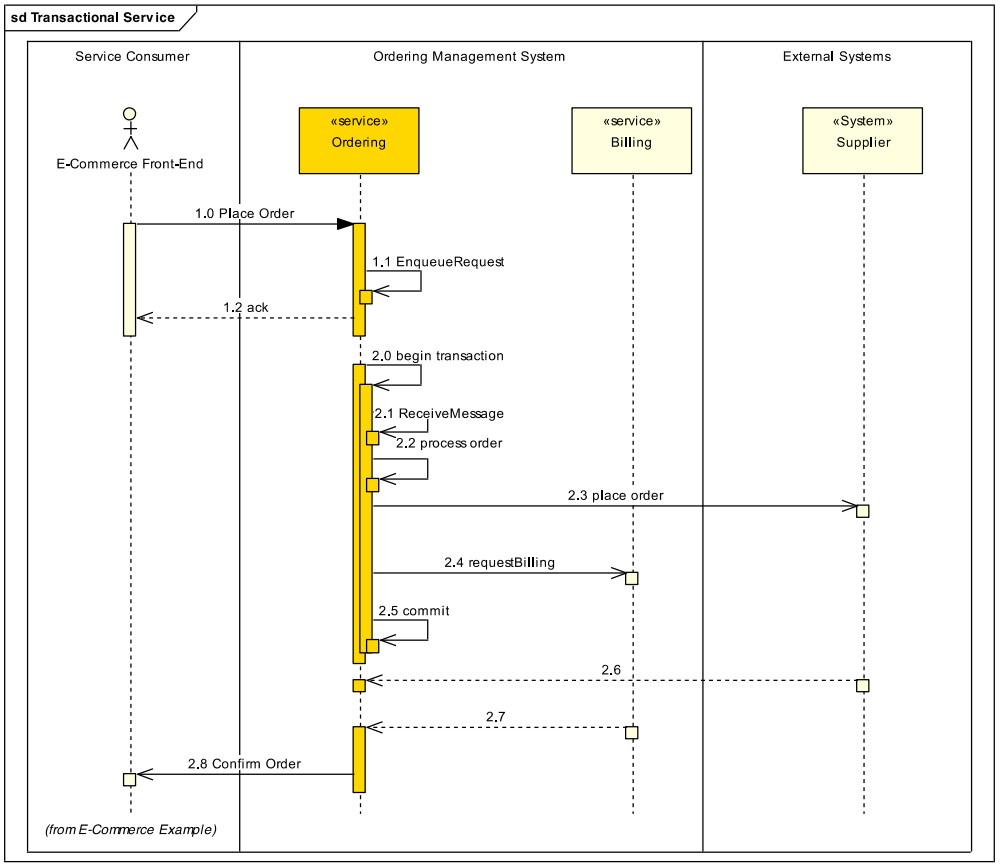


Figure 5.12 Sample message flow in an E-Commerce scenario (talking to an Ordering Service). The front-end sends an order to an ordering service which then orders the part from a supplier and asks a billing service to produce bill the customer. Note that all the actual handling of the Place Order message is done within a single local transaction.

Fine, so what's the problem? There are 2 major problems lurking here. Consider what will happen if instead of committing the internal transaction at step 2.5 the Ordering service decides to abort its (internal) transaction? Also consider how would the ordering service go about getting some commitment from the other services so that it would b able to continue its work based on that commitment. For instance, we may want to get a confirmation from

the supplier that she secured the items we ordered, before we confirm the order to our customer

5.4.1 The Problem

The obvious answer to the two problems mentioned above is to extend or flow the internal transaction which the ordering service already has into the other services. This "extended transaction" is known as a "distributed transaction". Using distributed transactions, the ordering service would have call both the billing service and the supplier's system as part of a single transaction and if all the services agree to commit the whole transaction is committed and completed together. This sounds really, really great (really ☺), we even have the technology to do that, which, by the way, predated SOA by many years.

But, and there's always a but… , what if the supplier can only complete their part of the transaction after a senior manager will authorize the deal ? Can we hold all our internal locks waiting for that manager to return from his vacation in the Bahamas sometime next week? Probably not… Even more so, if this supplier also happens to be a competitor. Now, they might prolong the transactions just to put us out of business - after all we hold locks on our internal resources while we wait for them to complete the transaction. The specific scenario I painted might be too farfetched but the point is that we can't make assumptions on how other services operate. This is especially true for services we don't own. There are additional reasons not to do cross-service transactions and you can read about them in detail in the Cross-Service Transactions anti-pattern (in chapter 8).

Even if you think that cross-service transactions are not problematic as a concept. You would probably agree that long transactions are not very good. So the more conversational the interaction between the services gets the more we need to think about alternatives atomic transactions. Again, if we look at the scenario in figure 5.12 we currently have 2 messages going out from the ordering service – which might be borderline in terms of number of interactions. However business processes can sometimes involve much more elaborate conversations.

A lot of messages flowing back and forth between services is not recommended as it increase latency and chances of failure. Nevertheless, few and sparse interactions are not realistic either. Services rarely live in complete isolation, after all, as mentioned in chapter 1, interoperability is one of the reasons we went with SOA in the first place. This mean we need to have a way to handle complex service interactions in a reliable way – without bundling the whole thing in one lengthy atomic transaction.

To sum both the problems we've see thus far, what we want to know is :

How can we reach distributed consensus between services without transactions?

I think by now it is clear that using a single transaction is not an option. If all the services involved are under your control you might want break the long process into multiple steps

and run each step in its own transaction. Smaller distributed transactions are definitely a step in the right direction. But we are still bound by cross-service transaction –and , because everything is not bounded by one single transaction we have problems like canceling the effect of a first step if something failed in the third or fourth one.

Another option is to try to model our contract so that we will never need this kind of complex interactions. We can minimize interactions between services if we increase the granularity of the services. However, there's also a limit to how large we want our services to be – we don't want to end up with a single monolith service that does everything. As we'd objects services need to be cohesive and adhere to the Single Responsibility Principle. When we do that we can contain some interactions within the service boundary but we still need to handle cross service interactions to implement business processes.

The option we are left with is to break the service interaction, that is our business process, to a set of smaller steps and model that into a long running conversation between the services.

5.4.2 The Solution

The Saga interaction pattern is about providing the semantics and components to support the long running conversation mentioned at the end of the previous section.

> **Implement the Saga pattern and break the services interaction i.e. the business process, to multiple smaller related business actions and counteractions. Coordinate the conversation and manage it based on messages and timeouts.**

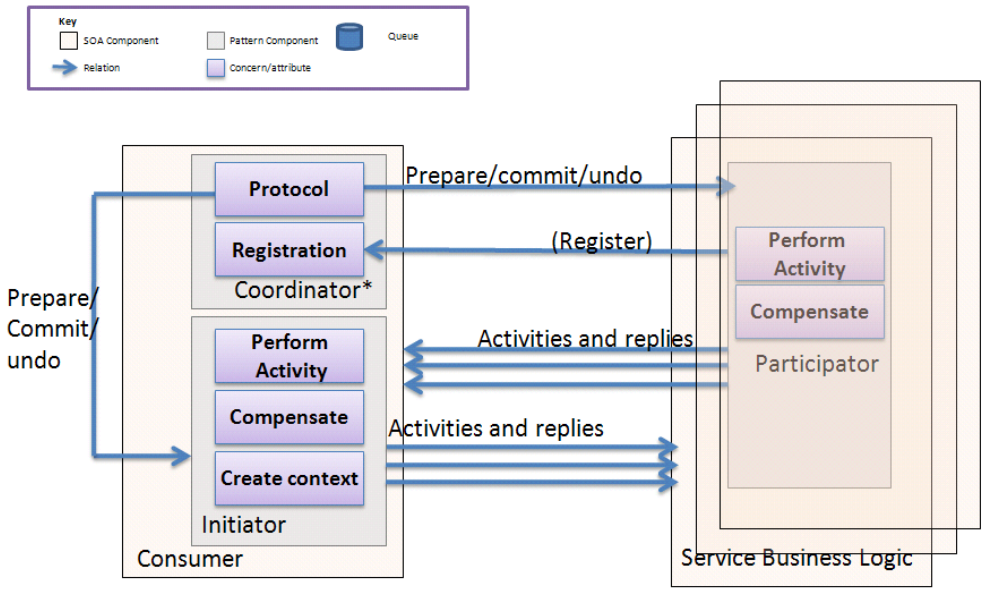


Figure 5.13 The Saga Pattern. A service consumer and one or more services hold a long running conversation within a single context (a Saga). Once the parties reach some consensus the conversation is committed. If there are problems during the conversation and the interaction is aborted. The involved parties perform corrective steps (compensations) . Note that the coordinator may be a component on its own (external to the consumer).

Hector Garcia-Molina and Kenneth Salem defined the term Saga back in 1987 as a way to solve the problem of long lived database transactions. Hector and Kenneth described a Saga as a sequence of related small transactions. In a Saga the coordinator (database in their case) makes sure that all of the involved transactions are successfully completed. Otherwise, if the transactions fails the coordinator runs compensating transactions to amend the partial execution. What made sense for databases makes even more sense for service interactions in SOA. We can break a long service interaction into individual actions or activities and compensations (in case of faults and errors)

The first component we notice is the initiator. The initiator triggers the Saga pattern by creating the context, the reason for the interaction. It then asks one or more other services (participators) to perform some businesses activities. The participators can register for coordination (depending how formal the Saga implementation is). The participants and initiator exchange messages and requests until they are reach some agreement or they are ready to complete the interaction. This is when the coordinator requests all the participant (including the initiator) to finalize the agreement (prepare) and commit.

If there was a problem either in during the interaction or the final phase the activities that occurred have to be undone. In regular ACID transactions you can rollback – however in a Saga you have to perform a counteraction, called compensation, which contrary to Newton's law[2], may not be the exact opposite of the activity that has to be undone. For instance if the result of the original activity the service crossed some threshold it may not wish to undo the action it took. Another example is that cancelling the action may require something form the service(s) that requested the action in the first place (e.g. cancellation fee) or that too much time has passed which makes it impossible to undo the effect. If we try to look at an example from the real world- if a result of a Saga was to launch the space shuttle, a compensation would be to abort the mission and return the shuttle home – but you can't just pull it back into the launch pad.

The Saga pattern is sometimes referred to as "Long Running Transaction". It is true that you can conceptually think of a Saga as a single logical unit of work and that it does make use of transaction semantics. However a Saga doesn't really adhere to the transaction tenets like atomicity or isolation – mostly because the interaction is distributed both in time and in space. For instance when you call a compensation it might be too late to undo the original action so that either or it might have consequences like cancellations fees or partial deliveries. I think the term Saga better reflects the fact the interaction is lengthy and that the messages are related.

Let's take a look at a how the interaction of the ordering scenario we presented in figure 5.12 above might look like when we utilize the Saga interaction pattern. Diagram 5.14 below demonstrate a scenario where the supplier is out of stock for the ordered items. In this case both the ordering and billing need to be canceled. We also need to notify the front-end that there was a problem and to let the supplier know we closed the interaction.

Using the Saga pattern, all the services involved (Ordering, Billing and the Supplier's service) notify their state. For instance the supplier sends a fault message to let the ordering service know it had a problem processing its request. When the coordinator component inside the Ordering service gets the fault message it requests the other parties, i.e. the ordering service itself and the billing service to compensate and once that done it notifies the supplier that the interaction completed handling the fault.

Also note that notifying the front-end about the failure is done during the compensation of the ordering service. It is not a task of the coordinator.

[2] Newton's 3rd law of motion: For every action there is a equal and opposite reaction

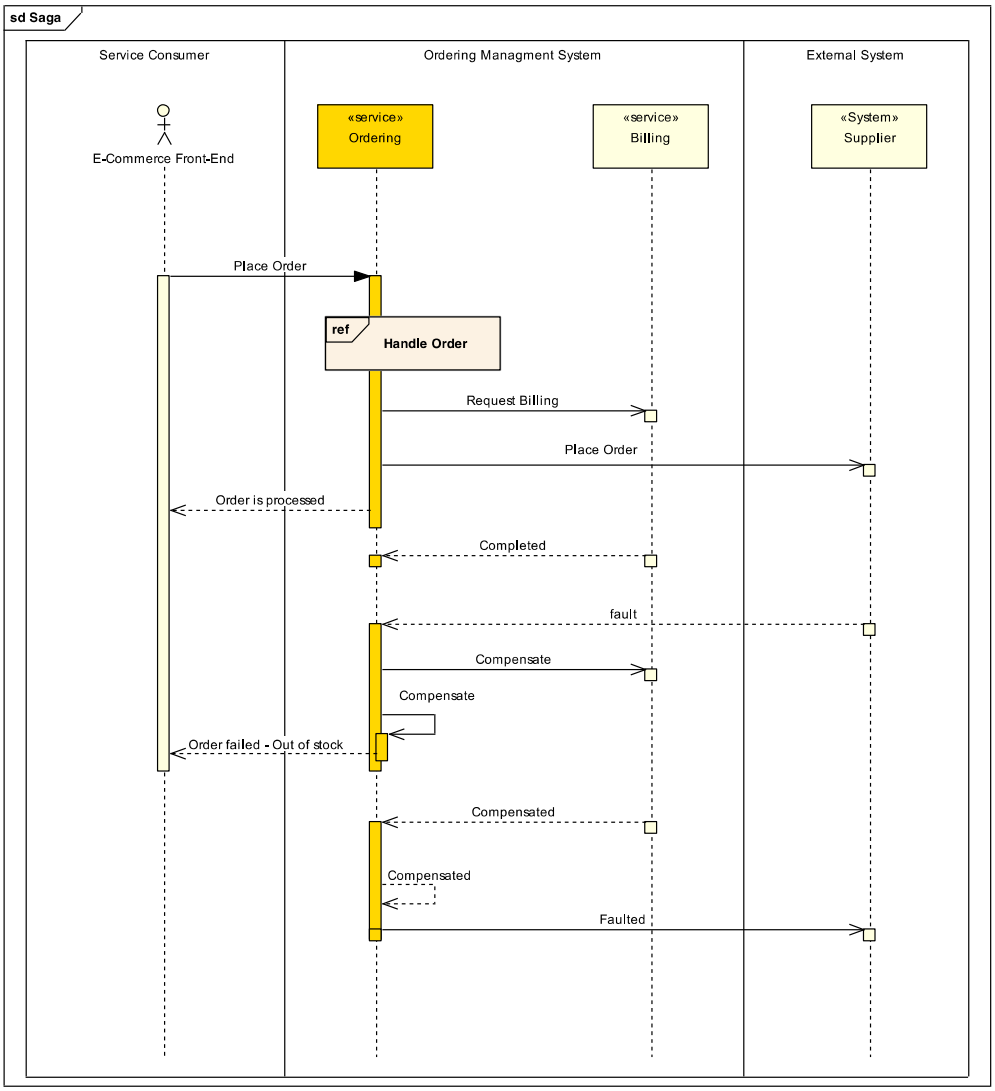


Figure 5.14 E-Commerce scenario from figure 5.12 remodeled using the Saga pattern. The interaction with the billing and the supplier is now coordinated in a saga. And the ordering service can handle problems in a more robust way by canceling the order and notifying the front-end instead of hoping for the best

The interaction above has the service consumer and services control the interaction internally. One good option to do this is to utilize the Workflodize pattern (see chapter 2) so

that each service holds a workflow internally which follows the sequence and different paths of the interaction. Other patterns related to the Saga pattern are correlated messaged (section 5.3) ad Reservation pattern (chapter 6)

Another approach you can take to implement the Saga pattern using an external coordinator for the conversation– see the Orchestrated Choreography pattern in chapter 7 for more details. The semantic difference between an internally coordinated Saga and an externally coordinated one is that the services involved in the first try to come to a mutual consensus while the services in the latter are driven to created a larger whole.

The main effort behind the Saga pattern is to decide on the business activities and compensations. You can use techniques such as Business Process Modeling (BPM – see also Orchestration in chapter 7) to form a good basis of what these activities might be.

Even though the main effort in implementing the Saga pattern is the business side, i.e. modeling business processes and activity that would support long running conversation, there are also a few technological aspects that have to do with the messages and protocol – let's take a look at them.

5.4.3 Technology Mapping

At the minimum the Saga pattern requires you to add compensation messages to any state altering message that can participate in a Saga. Again, it is important to emphasize that the compensation may not be able to undo the original activity – but it does have to try to minimize the effects of the activity. The innards of the processing of the compensation messages varies depending on what needs to be done to cancel the effect of the original message. Note that it is usually better to set statuses to cancelled rather than delete, especially at the database level, since the original action might have triggered other business processes and actions. For instance, if as a result of a message you added an order, another service might have produced a bill. Chances are that billing also occurred within the same Saga, but you might not know or control that within the ordering service. Making a change that leaves traces behind it (like cancel) is better than a delete since it also allows resolving problem manually if the need arises. Note that in some industries like banking you are required by law to register cancellations as new change rather than actually delete or amend (see also "accountants don't use erasers" by Pat Helland in further reading)

Another message type that is important for a Saga is failure message. When you have a simple point to point interaction between services the reply or reaction a called services sends is enough to convey the notion of a problem, the calling service consumer which understands the service's contract can understand that something is amiss and act accordingly. When you implement the Saga pattern however, you have the possibility of more than two parties and you also have a coordinator. The coordinator is not as business aware as the service's business logic but it does define control messages in order to understand the status of the interaction.

As you probably know (or at least notices by now) web-services are considered the primary technology for implementing SOAs and the Saga pattern is not different. The WS-* stack of protocol has produced the WS-BusinessActivity as part of WS-Coordination.

WS-BusinessActivity has two variants one which is a little more ordered and the other which is a little more loosely-coupled – with the cost being increased chances for compensation. The first is Business Agreement with Coordinator Completion – where the coordinator decides and notifies the participants when to complete and the Business Agreement with Participant Completion, where the participants decide when they complete their roles within the activity.

WS-BusinessActivity defines an orderly protocol and states for both the participating services and the coordinator. WS-BusinessActivity defines two coordination types – one, called AtomicOutcome where all the participants have to all close (commit) or compensate and another called MixedOutcome where the coordinator treats each participant separately. Additionally WS-BusinessActivity defines two protocols one where the coordinator decides when the participant fulfilled their share of the business process and one which is more loosely coupled, where each participant decides when it has finished its part of the process. For instance, Figure 5.15 below shows the state transitions for a participating service using the WS-BusinessActivity with participant completion.

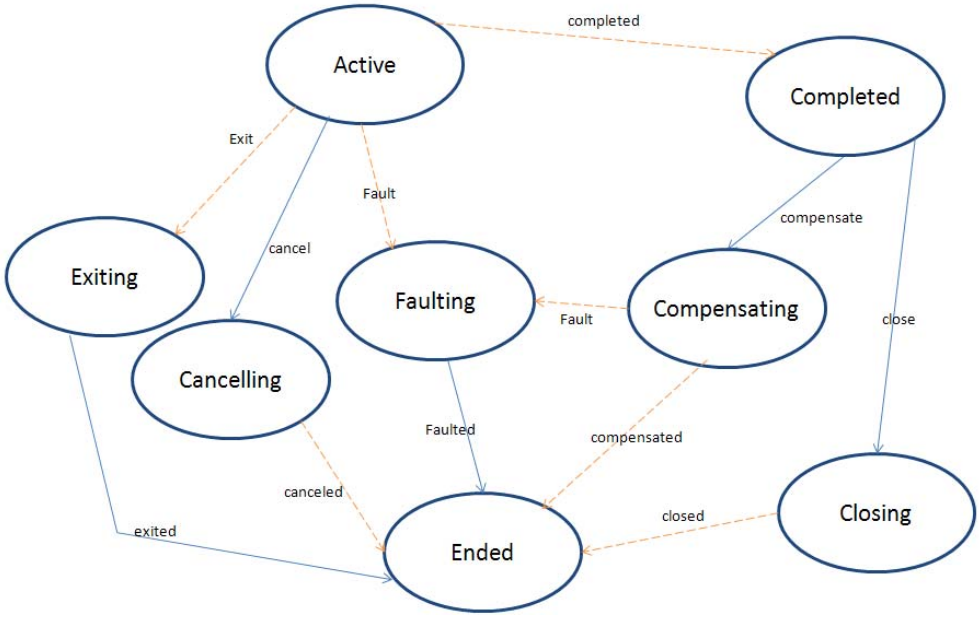


Figure 5.15 State diagram from the point of view of a participating service using the completion by participants variant of the WS-BusinessActivity protocol. The state transitions can be either the result of decisions by the service (the dotted lines) or by messages from the coordinator (the full lines)

Another important technology option for implementing the Saga pattern is to use BPEL (Business Process Execution Language) or it WS-* implementation known as WS-BPEL (or BPEL4WS in previous versions). Additionally you can also use a non-BPEL compliant orchestration engine. In any even these technology mapping fall under the external coordinator mentioned above and are covered in more depth as part of the Orchestration pattern in chapter 7.

5.4.4 Quality Attributes

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. The scenarios are used to describe the architectural requirements in a way that allows evaluating architecture to see if it answers them. Another use for the scenarios is to design an architecture – or from the perspective relevant here - as a way to identify situations where a pattern is applicable.

The main reason to employ the Saga pattern is to increase the integrity of the system. As I already mentioned in the sections above, transactions are problematic when it comes to distributed environment in general and even more so when using SOA. Nevertheless we still want to be able to coordinate the behavior of services and get meaningful interaction. Be letting us coordinate the behavior and failure handling we introduce a reliable, predictable long-running conversations.

Another aspect of integrity which is a reason to use the Saga pattern is to increase the predictability. In a distributed environment it is relatively hard to know what the outcome will be, this is especially true if you use other patterns like Inversion of Communications (section 5.3). The Saga patterns allows introducing some control into the interaction and verify that the outcome of a complex interaction be along known paths (completed or compensated).

Lastly, the outcome of increased predictability is also increased correctness. Knowing how the system is going to behave it is easier to construct system tests to verify that the desired outcome indeed happens

Table 5.6 Saga pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using the Decoupled Invocation pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Integrity	Correctness	Under all conditions, an order processed by the system will be billed
Integrity	Predictability	Under normal conditions, the chances of a customer getting billed for a cancelled order shall be less than 5%

Reliability	Handling failure	Resuming from a communications disconnection , all the processes that were interrupted shall remain consistent

Writing compensation logic is relatively complicated as the timeline advances the number of changes in the service can get rather large, which makes it harder to get the predictability when you try to undo an early change. One way to try to cope with that is to implement the Reservation pattern which you can read about in the next chapter

5.5 Summary

One distinct characteristic of all the patterns in this chapter is that none of them are new. All the interaction patterns predate SOA by many years. Nevertheless, I've spent more than 30 pages discussing them with you, instead of, for example, just pointing you to the (excellent) Enterprise Integration Patterns book, which covers these patterns as well. The reason for this is that although these patterns seem relatively simple and well known, each has some aspects that makes them a little complicated when we try to implement and adhere to the SOA principles:

- Request/Reply – Talks about synchronous communications. however in SOA it is better to do document based interactions. Vesrus the RPC based interaction which is the norm in "traditional" distributed architectures for synchronous communications
- Request/Reaction – Asynchronous communications, again a simple pattern, but it can be tricky to implement it when you use consumers that don't supports callbacks
- Inversion of Communications – Eventing, but with a few twists such as implementation on transports that don't actually support eventing. Another interesting aspect is providing event streams. Lastly there was the issue of not handling subscriptions (at least not directly)
- Saga – Transactions, but not really, it is actually a bad approach to implement ACID transactions between services. Instead we take a mechanism for reaching distributed consensus which was originally defined for distributed databases and apply it to SOA

The next two chapters will look at less basic interaction patterns – some of them are complimentary to the patterns discussed here e.g. the Reservation pattern in chapter 6, which complements the Saga pattern or the aggregated reporting in chapter 7 that uses the Inversion of Communications pattern. The other patterns have to do with the interactions and aggregations aspects beyond the underlying message exchange patterns e.g. the Consumer/Service pattern in chapter 6.

5.6 Further Reading

This section provides links to resources (web or otherwise) for all the technologies discussed in this chapter.

Table 5.7 resources for further reading on topics covered in this chapter.

Topic	Resource name/link	Why
Inversion of Communications Pattern	Bridging the gap between BI & SOA http://www.infoq.com/articles/BI-and-SOA	Shows an application of the Inversion of communications pattern (as well as Aggregated Reporting)
All patterns in chapter 5	Enterprise Integration Patterns – Gregor Hohpe & Bobby Woolf Orielly	Discuss fundamental integration pattern in general context many of which are applicable to SOA as well.
Document Centric interactions	http://code.google.com/apis/gdata/overview.html	Google's GData protocol is an example for a document centric protocol for interacting with services.
Inversion of Communications Pattern	http://www.eecs.harvard.edu/~mdw/proj/seda/	Combining Inversion of Communications with Parallel Pipelines pattern gives a SOA implementation of SEDA

Saga	http://blogs.msdn.com/b/pathelland/archive/2007/06/14/accountants-don-t-use-erasers.aspx	Pat Helland explaining the merits of retaining prior states

5.7 Bibliography

Hector Garcia-Molina, Kenneth Salem: International Conference on Management of Data: Proceedings of the 1987 ACM SIGMOD international conference on Management of data, San Francisco, California, United States
Pages: 249 - 259
Year of Publication: 1987
ISSN:0163-5808

http://elementallinks.typepad.com/bmichelson/2006/02/eventdriven_arc.html

Patrick Sauter, Ingo Melzer, http://www.ingo-melzer.de/papers/kivs05.pdf, A Comparison of WS-BusinessActivity and BPEL4WS Long-Running Transaction, KIVS 2005, Kaiserslautern, Germany Feb 2005.

http://www.onjava.com/pub/a/onjava/2005/07/27/axis2.html Web Services Messaging with Apache Axis2: Concepts and Techniques

http://www.gartner.com/DisplayDocument?doc_cd=141940 Advanced SOA for Advanced Enterprise Project, Gartner
Fielding Dissertation

6

Service Consumer Patterns

The previous chapter focused on the basics of service interactions – the message exchange patterns. Chapter 6 also focuses on the interactions of services with their consumers but covers a wider angle, looking at useful patterns to support these interactions. The chapter's focus is demonstrated, as usual, on the SOA components model (Figure 6.1) where the highlight is on the Service Consumer.

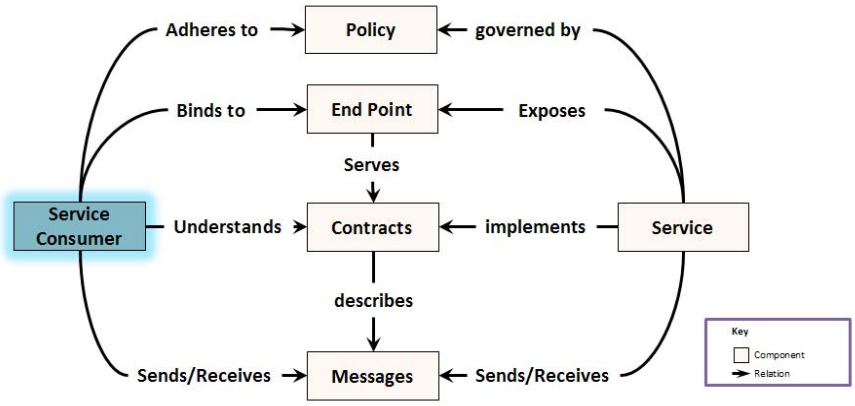


Figure 6.1 Chapter 6 focus is about connecting Services with Service consumers in the levels and layers beyond the basic message exchange patterns

It is important to note that Service consumers are not necessarily other services (though that is a possibility as well). For instance one particular consumer is the User Interface (UI). It is important to talk about connecting the UI as SOA in itself doesn't really pay attention to the needs of UIs. SOA separates business concepts into different services. Users working

against a UI want a unified view. The chapter takes a look at UI integration patterns with patterns like Client/Server/Service that talk both about legacy and thin client integration, and the Service/Consumers that talk about rich client integration and a few other patterns. Another type of non-services consumers are non-SOA applications, we'll take a look at connecting them in with the Service Agent pattern. Other patterns in this chapter include the Reservation pattern that enhances the Saga pattern and Service bus that is a service infrastructure pattern to connect services.

Table 6.1 summarizes the patterns discussed in this chapter and the challenges that they address.

Table 6.1 list of patterns covered in chapter 6.

Pattern name	Problem address
Reservation	How can we efficiently provide a level of guarantee in a loosely coupled manner while maintaining services' autonomy and consistency?
Composite Frontend	How do you we interact with multiple services, get an integrated, cohesive user interface and still preserve SOA principles and modularity benefits?
Client/Server/Service	How do you connect an SOA to user interfaces where integration is problematic e.g. the client side is not SOA aware or it uses incompatible technologies

The first pattern we will look at is the Reservation pattern, which is closely related to the Saga pattern discussed in the previous chapter. The Reservation chapter also exists in its own right to allow a service to give partial commitment when it interacts with its consumers.

6.1 Reservation Pattern

When you use transactions in "traditional" n-tier systems life is relatively simple. For instance, when you run a transaction and an error or fault occurs you abort the transaction and easily rollback any changes, getting back your system-wide consistency and peace of mind. The reasons this is possible is that a transaction isolates changes made within it from the rest of the world. One of the base assumptions behind Transactions is that the time that elapses from the beginning of the transaction until it ends is short. Under that assumption we can afford the luxury of letting the transaction hold locks on our resources (such as databases) and prevent changes by others while the transaction is in progress. Transactions provide four basic guarantees – Atomicity, Consistency, Isolation and Durability, usually remembered by their acronym - ACID.

Unfortunately, in a distributed world (SOA or otherwise), it is rarely a good idea to use atomic short lived transactions (see the Cross-Service Transactions anti-pattern in chapter 10 for more details). Indeed, the fact that cross service transactions are discouraged is one of the main reasons we'd consider using the Saga pattern in the first place.

One of the obvious shortcomings of Sagas is that you cannot perform rollbacks. The two conditions mentioned above, locking and isolation do not hold anymore so you cannot provide the needed guarantee. Still, since interactions, and especially long running ones, can fail or be canceled Sagas offer the notion of Compensations. Compensations are cool; we can't have rollbacks so instead we will reverse the interaction's operation and have a pseudo rollback. If we added one hundred (dollars/units/whatnot) during the original activity we'll just subtract the same 100 in the compensation. Easy, right? Wrong – as you probably know, it isn't easy.

6.1.1 The Problem

Unfortunately, there are a number of problems with compensations. These problems come from the fact that, unlike ACID transactions, the changes made by the Saga activities are not isolated. The lack of isolation means that other interactions with the service may operate on the data that was modified by an activity of other sagas, and render the compensation impossible. To give an extreme example, if a request to one service changes the readiness status of the space shuttle to "all-set" and another service caused the shuttle to launch based on that status, it would be a little too late for the first service to try to reverse the "all-set" status now that the "bird has left the coop". A more down to earth (pardon the pun) business scenario is any interaction where you work with limited resources e.g. ordering from a, usually limited, stock.

Consider the scenario in figure 6.1. A customer orders an item. The ordering service requests the item from the warehouse, as it wants to ship the item to the customer (probably by notifying another service). Meanwhile on the warehouse service the item ordered causes a restocking threshold to be hit that triggers a restocking order from a supplier. Then the customer decides to cancel the order – now what?

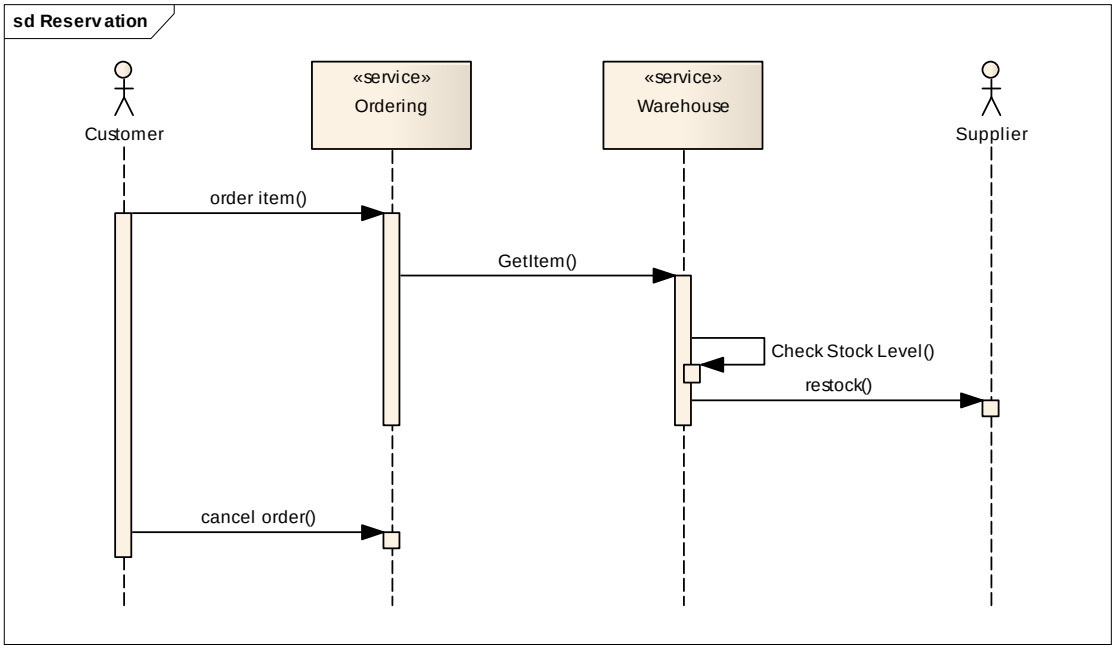


Figure 6.1 Chapter 6 focus is about connecting Services with Service consumers in the levels and layers beyond the basic message exchange patterns

Should the restocking order be cancelled as well? Can it be cancelled under the ordering terms of the supplier? Also a customer requesting the item between the ordering and cancellation might get an out of stock notice which will cause him to go to our competitors. This can be especially problematic for orders which are prone for cancellations like hotel bookings, vacations etc.

Another limitation of compensations and the Saga pattern itself, for that matter, is that it requires a coordinator. A coordinator means placing trust in an external entity, i.e., outside (most) of the services involved in the saga, to set things straight. This is a challenge for some of the SOA goals as it compromises autonomy and introduces unwanted coupling to the external coordinator.

The question then is

How can we efficiently provide a level of guarantee in a loosely coupled manner while maintaining services' autonomy and consistency?

We already discussed the limitations of compensations, which of course is one of the options to solve this challenge. Again, one problem is that we can't afford to make mini changes since we will then be dependent on an external party to set the record straight. The other

problem with compensations is that we expose these "semi-states" – which are essentially, the internal details of the services, to the out-side world. Increasing the footprint of the services' contract, esp. with internal detail, makes the services less flexible and more coupled to their environment (See also the white box services anti-pattern in chapter 10)
We've also mentioned that distributed transactions is not the answer since they both lock internal resources for too long (a Saga might go on for days..?) as well as put excess trust on external services which may be external to the organization. So what's the solution?

6.1.2 The Solution

This seems like a quagmire of sorts, fortunately, real life already found a way to deal with a similar need for fuzzy, half guarantees – reservations! (See figure 6.2)

Implement the Reservation pattern and have the services provide a level of guarantee on internal resources for a limited time

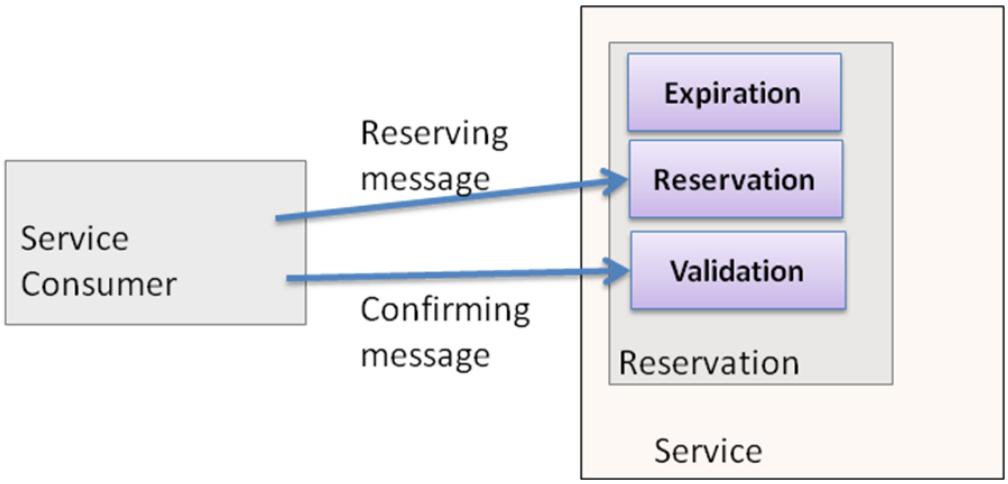


Figure 6.2 The Reservation pattern. A service that implement reservation consider some messages as "Reserving" in which it tries to secure an internal resource and sends confirmation if it succeeds. When a message considered as "confirming" the service validate the reservation still holds. In between the service can choose to expire reservation based on internal criteria

The Reservation pattern means there will be an internal component in the service that will handle the reservations. Its responsibilities include:

- **Reservation** - making the reservation when a message that is deemed "reserving" arrives. For instance when an order arrives, in addition to updating some durable storage (e.g. database) on the order it needs to set a timer or an expiration time for the order confirmation alternatively it can set some marker that the order is not final.

- **Validation** – making sure that a reservation is still valid before finalizing the process. In the ordering scenario mentioned before that would be making sure the items designated for the order were not given to someone else.
- **Expiration** – marking invalid reservation when the conditions changed. E.g. if a VIP customer wants the item I reserved, the system can provision it for her. It should also invalidate my reservation so when I finally try to claim it the system will know it's gone. Expiration can also be timed, as in, |we're keeping the book for you until noon tomorrow"

Reservations can be explicit i.e. the contract would have a ReserveBook action or implicit. In case of an implicit order the service decides internally what will be considered as Reserving message and what will be considered as confirming message e.g. an action like Order will trigger the internal reservation and an action like closing the saga will serve as the confirming message. When the reservation is implicit the service consumer implementation will probably be simpler as the consumer designers are likely to treat reservation expiration as "simple" failures whereas when it is explicit they are likely to treat the reservation state.

Reservations happen in business transactions world-wide every day. The most obvious example is booking a hotel. You send in a request for a room (initiate a saga) saying you'd arrive on a certain date, say for a conference, and check out on another (perform an action within the saga). The hotel says ok, we have a room for you (reservation) – provided you confirm your arrival by a set-date (limited time). Even if everything went well, you may still arrive at the hotel, only to find out your room has been given to another person (limited guarantee). The idea of the reservation pattern is to copy this behavior to the interaction of services so that services that support reservations offer a sort of "limited lock" for a limited time and with a limited level of guarantee. Limited level of guarantee, means that like real life, services can overbook and then resolve that overbooking by various strategies such as first come, first served; VIP first served etc

It is easy to see Reservation applied to services that handle "real-life" reservations as part of their business logic, such as a ordering service for hotels (used in the example above) or an airline etc., However reservations are suitable for a lot of other scenarios where services are called to provide guarantees on internal resources. For instance, in one system I built we used reservations as part of the saga initiation process. The system uses the Service Instance pattern (see chapter 3) where some services are stateful (the reasons are beyond the scope of this discussion). Naturally, services have limited capacity to handle consumers (i.e. an instance can handle n-number of concurrent sagas/events). This means that when a saga initialized all the participants of the saga needs to know the instances that are part of the saga. As long as a single service instance initiates sagas everything is fine. However, as illustrated in figure 6.3, when two or more services (or instances) initiate sagas concurrently they may (and given enough load/time they will) both try to allocate the same service instance to their relative sagas. In figure 6.3 we see that both Initiator A and Initiator B want to use Participant A and Participant B. Participant A has a capacity of 2 so everything

is fine for both Initiators. Service B, however, has limited capacity so at least one of the Sagas will have to fail the allocation, i.e. not start.

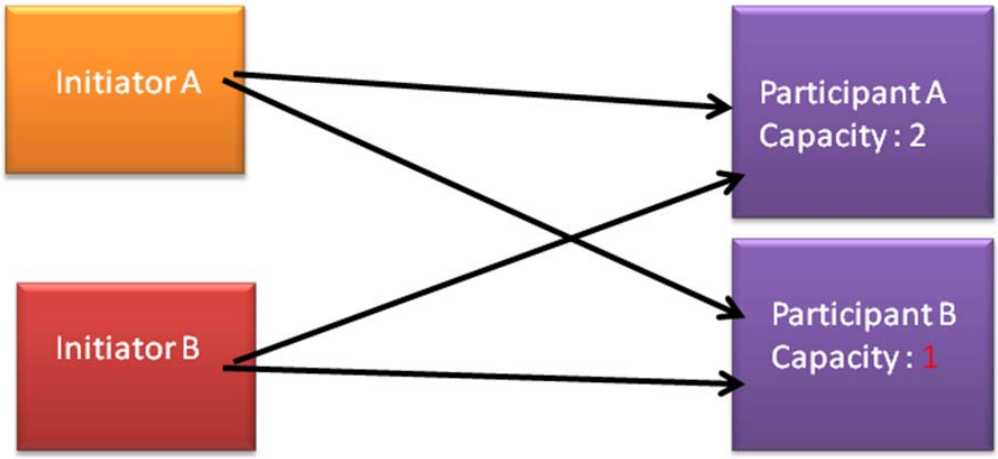


Figure 6.3 : Sample for a situation that can benefit from the reservation pattern

The reservation pattern enabled us to manage this resource allocation process in an orderly manner by implementing a two pass protocol (somewhat similar to a two phase commit). The initiator asks each potential participant to reserve itself for the saga. Each participant tries to reserve itself and notify back if it is successful – so in the above scenario, A would say yes to both and B would say yes to one of them. If the initiator gets an OK from all the involved services (within a timeout) it will tell all the participants the specific instances within the saga (i.e. initiate it).

The participants only reserve themselves for a short period of time. Once an internally set timeout elapse the participants remove the commitment independently.

> As a side note, I'll just say that the initiator and other saga members can't assume that the participant will be there just because they are "officially" part of the saga and the system still needs to handle the various failure scenarios. The Reservation pattern is used here only to help prevent over allocation and it does not provide any transactional guarantees.

A reservation is somewhat like a lock and thus it "somewhat" introduces some of the risks distributed locks presents. These risks aren't inherent in the pattern but can easily surface if you don't pay attention during implementation (e.g. using database locks for implementation).

- The first risk worth discussing is deadlock. Whenever you start reserving anything, esp. in a distributed environment you introduce the potential for deadlocks. For instance if both participants had a capacity for single saga, initiator A contacts participant A first and participant B next and initiator B used the reverse order – we would have had a deadlock potential. In this case there are several mechanisms that prevent that deadlock. The first is inherent to the Reservation pattern, where the participants release the "lock" themselves. However, for example, if there is a retry mechanism to initiate the sagas (as both would fail after the timeout) and the same resources will be allocated over and over there may be a deadlock after all
- Another risk to watch out fir when implementing Reservations is Denial of Service (whether maliciously or as an byproduct of misuse). DoS can happen from similar reasons discussed in the deadlock (i.e. if you incur a deadlock you also have a DoS). Another way is via exploiting the reservations by constantly re-reserving. Depending on the reservation time-out, regular firewalls might fail detecting the DoS so you may want to consider using a Service Firewall (chapter 4) to help mitigate this thread.
- Besides the risks discussed above, another thing to pay attention to is when you introduce Reservation because you are likely to add additional network calls. The system discussed above mention that when it introduce another call tell the Saga members which instances are involved in the saga.

In addition to the Service Firewall pattern, mentioned above, another pattern related to Reservations can be the Active Service pattern (see chapter 2). The Active Service pattern can be used to handle reservation expiration when implemented by timed.

NOTE: Sometimes it's better, resource-wise, to handle expiration passively and not actively, as we'll see looking at it's implementation options in the next section.

6.1.3 Technology Mapping

Unlike a lot of the patterns in this book, the Reservation pattern is more a business pattern than a technological one. This means there isn't a straight one-to-one technology mapping to make it happen. On the other hand, code-wise, the pattern is relatively easy to implement.

One thing you have to do is to keep a live thread at the service to make sure that when the lease or reservation expires someone will be there to clean up. One option is the Active Service pattern mentioned above. You can use technologies that support timed events that provide the "wakeup service" for you. For instance, if you are running in an EJB 3.0 server you can use single action timers i.e. timers that only raise their event once to accomplish this. The listing below shows a simple code excerpt to set a timer to go off based on time received in the message. Other technologies provide similar mechanism to accomplish the same effect.

Listing 6.1 setting a timer event (using JBOSS)

```
public class TimerMessage implements MessageListener {

  @Resource
  private MessageDrivenContext mdc;
    .
    .
    .
  public void onMessage(Message message) {
     ObjectMessage msg = null;
     try {                                                              #1
           if (message instanceof ObjectMessage) {
          msg = (ObjectMessage) message;
           TimerDetailsEntity e = (TimerDetailsEntity) msg.getObject();
           TimerService timerService = messageDrivenCtx.getTimerService();

          // Timer createTimer(Date expiration, Serializable info)   #2
          Timer timer = timerService.createTimer(e.Date, e);
         }
     } catch (JMSException e) {
        e.printStackTrace();
        mdc.setRollbackOnly();
     } catch (Throwable te) {
        te.printStackTrace();
     }
  }
.
.
.
```

(Annotation) <#1 vanilla code to process a message and get the interesting entity out of it >
(Annotation) <#2 set the single action timer based on the info in the message we've just got>

Timer based cancellation, as described above, might be overkill if the reservation implementation is simple. The Reservation, in the listing below implemented in C#, is used by the participants, which was discussed in the Saga and reservation sample in the previous section.

Listing 6.2 Simple in-memory, non-persistent reservation

```
public Guid Reserve(Guid sagaId)
        {
            try
            {
                Rwl.TryWLock();
                var isReserverd = Allocator.TryPinResource(localUri,
sagaId);
                if (!isReserverd)                                       #1
                    return Guid.Empty;

               //Some code to set the expiration                     #2
                return sagaId;                                          #3

            }
```

```
            finally
            {

               Rwl.ExitWLock();
            }

   }
```

(Annotation) <#1 The allocator is a resource allocation control, which manages, among other things, the capacity of the service. If we didn't succeed in marking the service as belonging to the Saga, we can't allocate the service to the specific Saga>
(Annotation) <#2 Here is where we need to add code to mark when the reservation expired, the previous example (6.1) used timers , we'll try to do something different here see listing 6.3>
(Annotation) <#3 successful reservation returns the SagaId this assures the caller that the reply it got is related to the request it sent – a simple Boolean might be confusing >

Since the Reservation in listing 6.2 does not involve heavy service resources (like, say, a database etc.), we can implement a passive handling of reservation expiration, which will be more efficient than a timer based one. The listing below shows both a revised reservation implementation, which removes timeout reservation before it commits. Note: that using this code an expired reservation can still be used if no other reservation occurred in between or the capacity of the service is not exceeded.

Listing 6.3 passive reservation expiration handling (added on top of the code from listing 6.2)

```
private readonly TimeSpan MAX_RESERVERVATION = new TimeSpan(0, 0, 0, 1, 0);
.
.
public Guid Reserve(Guid sagaId)
        {
            try
            {
                Rwl.TryWLock();
                RemoveExpiredReservations();                              #1
                var isReserverd = Allocator.TryPinResource(localUri,
sagaId);
                if (!isReserverd)
                    return Guid.Empty;

                OpenReservations[sagaId] = DateTimeOffset.Now +
MAX_RESERVERVATION;                                                       #2
                return sagaId;

            }
            finally
            {

               Rwl.ExitWLock();
            }

        }
```

```
 private void RemoveExpiredReservations()
        {
            var reftime = DateTimeOffset.Now;
            var ids = from item in OpenReservations where item.Value <
reftime select item.Key;
            if (ids.Count() == 0) return;
            var keys=ids.ToArray();
            foreach (var id in keys)
            {
                OpenReservations.Remove(id);
                Allocator.FreePinnedResources(id);
            }

        }
```

(Annotation) <#1 Added a small method (RemoveExpiredReservations which also appears in the listing) to clean expired reservations. This method is ran everytime the service needs to handle a new reservation request and it cleans up expired reservations. Note that there is no timer involved, reservation are only cleaned if there is a new reservation to process>
(Annotation) <#2 Instead of a timer the reservation is done by marking down when the reservation will expire>

The code samples above show that implementing Reservation can be simple. This doesn't mean that other implementations can't be more complex. For example if you want/need to persist the reservation or distribute a reservation between multiple service instances etc., but at its core it shouldn't be a heavy or complex process.

Another implementation aspect is whether reservations are explicit or implicit. Explicit reservation means there will be a distinct "Reserve" message. This usually means there will also be a "Commit" type message and that the service or workflow engine that request the Reservation might find itself implementing a 2-phase commit type protocol, which isn't very pleasant, to say the least.

The other alternative is implicit where the service decides internally when to reserve and what conditions to commit the reservation and when to reject it. As usual the tradeoff is between simple implementation to the service and simple implementation for the service consumer.

6.1.4 Quality Attributes

As usual, we wrap up the pattern by taking a brief look at some business drives (or scenarios) that can drive us to use the reservation pattern. In essence, the main drive to reservation is the need for commitment from resources and since it is a complementary pattern to Sagas it also has similar quality attributes. As mentioned above, Reservation helps provide partial guarantees in long running interactions thus the quality attribute that point us toward it is Integrity.

Table 6.2 Reservation pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using the Reservation pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Integrity	Correctness	Under all conditions, failure receive payment within 5 business days will cancel the order and shipping
Integrity	Predictability	Under normal conditions, the chances of a customer getting billed for a cancelled order shall be less than 5%

Reservation is a protocol level pattern that involves exchange of messages between service consumers and services. The next couple of patterns take a look at a component that may actually need to use reservations when they talk to services – the user interface and how to tie it to services running at the backend.

6.2 Composite Frontend (Portal)

When we try to think about service consumers, the obvious candidates are, of course, other services. Nevertheless there are other software components that interact with services e.g. legacy systems, Non-SOA external systems or reporting databases. The Composite Frontend pattern deals with yet another type of service consumer – the User interface.

First let's just verify that User interfaces aren't in fact services. One reason user interfaces are not services is that they converge several business areas. if you want to enter an order you'd probably also want to lookup information about the customer, maybe you'd also want to browse the product catalog, look at open invoices. In addition to convergence, user interfaces deliver data rather than process it. User interfaces are data producers (actually there's one exception to that – where the UI is the front of a "human service" see orchestrated choreography pattern (in chapter 7) for more details.

Ok, so UIs aren't services, does it matter? Well, it does and the problem is not that UIs aren't services per se. The main challenge caused by user interfaces comes from their main difference: the aggregation or convergence of several services into a cohesive and useful UI.

6.2.1 The Problem

To better understand the challenges caused by user interfaces working with multiple services, let's consider an example with just a single point of friction.

In a project I worked on we've designed the C4ISR (Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance) system for an Unmanned Naval Patrol Vehicle (UNPV). One of the services in the system was dubbed "Common Operational Picture" or COP for short. The COP 's responsibility was to handle anything that is detected by sensors: ships, planes, whatever (There's technical jargon for all

that like targets, detections, tracks but that's not important here). One of the main UI representations of the COP was a map, see figure 6.4, which showed all the detections. Clicking on map icon, say a ship, presents some information the COP knows about it, such as id, nationality, course.

Figure 6.4 A simplified illustration of a front end for a "Common Operational Picture" service of a naval command and control system. We see a shore line and then using NATO symbology: 2 radars, a submarine and an a ship (the Unmanned Naval Patrol Vehicle)

The system had a few other services in addition to the COP, amongst them there was the "UNPV service". The UNPV service was responsible for anything related to the UNPV itself for instance, setting it on a navigation course, turning it around. The UNPV service had several UI screens to allow managing and monitoring these functions. Another responsibility of the UNPV service was to send its location to the COP (locations are the COP's responsibility, remember?) so back in the UI one of the icons of the map is that of the UNPV.

What happens when a user clicks on the UNPV icon on the map? Remember, while the desired outcome is to display a popup with options related to controlling the UNPV, the click is on the map, a control serviced by another service (the COP). In an Object Oriented system the UNPV might be a sub-class of a detection so it would accept the same event and respond a little differently (in a more specialized way). Here, however the COP and the UNPV are completely different services, developed by two different groups and maybe even two different companies.

We may be able to dismiss this specific example and just solve it with a specific solution: add an 'if' statement somewhere to call up the correct commands and to interact with the correct service(s). The problem, however, is that this example is just the tip of the iceberg. How do we handle security? Do we need to login for each service separately? How do we handle things that all the services need? SOA's premise is that we'd get a sort of lego-like enterprise where we can compose different business processes easily. Is there any way we can get that in the user interface? In summary:

How do you we interact with multiple services, get an integrated, cohesive user interface and still preserve SOA principles and modularity benefits?

One option is, as mentioned above, write specific client code. Using this approach "an application", is any specific composition of services. For the example above, the application would include the two service (COP and UNPV) and a UI that ties them together. The up-side of this approach is that each application delivers a consistent experience for the user. After all, a specific or tailored application can be made to be very cohesive. Additionally there are many tried and tested ways to build flexible UIs with a proper separation of concerns, e.g. using Model-View-Controller and its variations (in a multitude of rich client and web technologies) so we'd probably be able to reuse some of the UI-side logic even going from application to application. Nevertheless, we do lose on flexibility. For one, any service change that has UI aspects needs to be redone for each of its UI instances (applications). More so, since the UI specifically ties multiple services changes in one service may cause another to mal-function within the unified UI. We also lose on Composability, or the ability to replace services and to create new business flows (relatively) easy. Overall it's a bad option long-term but it can be made to work as a short term solution.

A related option is taking a similar approach of tying several services together but instead of integrating them on the client side we integrate the services together on the server side. This approach shares it pros and cons with the previous solution. However there are specific circumstances where it does make sense and you can read about them in the next pattern (section 6.5 Client/Server/Service)

Lastly, we have the option to have independent UI components per service. This would overcome the limitations we've mentioned above since each service's corresponding UI can evolve independently and you can just cram a as many of these as you like to create an application. Unfortunately, with all its benefits this is a non even an option here as, by definition, we won't have the mechanisms for UI components that work cross services. i.e. it won't actually solve problems like the one in the example and we can't get a cohesive UI.

6.2.2 The Solution

What we need, essentially, is a way to compose services together while keeping their respective autonomy on one hand and providing mechanisms to glue them together as a cohesive whole – That's what the Composite Frontend pattern is about:

Apply the Composite Frontend pattern to aggregate services while providing them unified client-side services like layout and theming as well as coordination services for client-side service integration

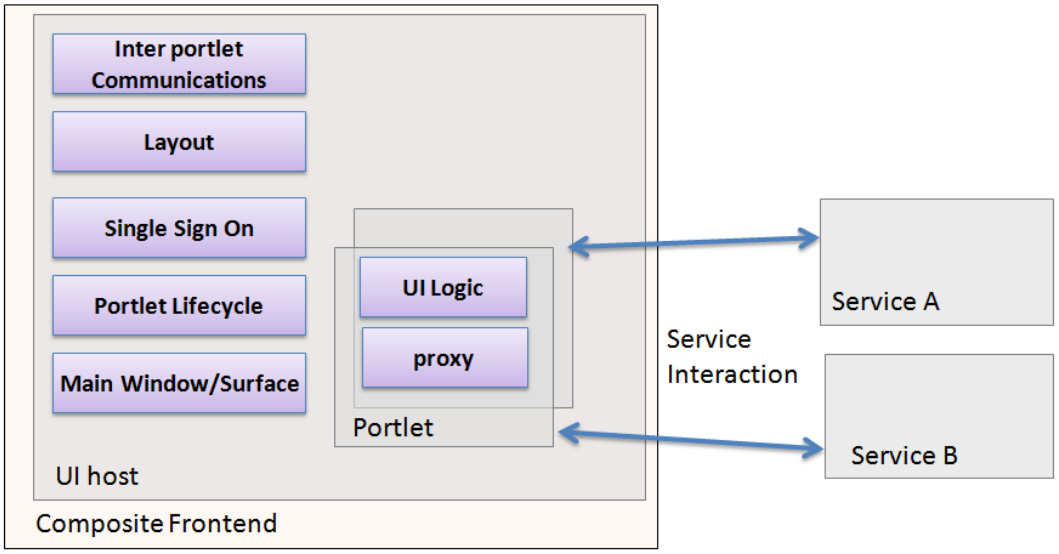


Figure 6.5 The Composite Frontend pattern. Each Service has a Portlet which is a Service Agent combined with a UI logic (most likely Model in a MVC UI pattern). The UI host provides services for the different potlets to weave them together into a coherent UI.

The Composite Frontend pattern is about taking the ideas (and sometimes the technologies) behind web portal and applying them to SOA services. Web portals provide unified access point that aggregates multiple web pages. They also provide single sign-one and personalization. SOA interfaces need that and more.

The composite front-end pattern is composed of two main components: the portlet and the host.

PORTLET

The portlets are the building blocks, "the composites" which are fused together to form the user interface. The portlets are made of at least two components. The user interface logic (views and controllers in MVC lingo). The second component, the service proxy (or

agent) is the more interesting one from an SOA perspective. The service proxy is a client-side representation of a service. The proxy serves as the model for the user interfaces components. It is usually recommended to have a single proxy per service -just like it is recommended for each service to maintain its own data store. Furthermore, *from a product management perspective it can be seen as part of the service itself*

HOST

The host is the "value-add" part of the composite frontend pattern. The host provides the glue that ties the different portlets into a cohesive whole. As such, the host provides several roles.

- provides the canvas or surface on which the portlets are displayed.
- controls the life-cycle of the portlets
- provides capabilities (avoiding the loaded term services...) like inter-portlet communications and single-sign-on.

Let's revisit the problem discussed in the previous section. We had a right-click on a ui component, which should have produced a context menu with options from two services. How would that with the composite frontend pattern? One option is that a click would be first intercepted by the host that would then dispatch it to any registered portlet. Another option illustrated in figure 6.6, is for the click to be intercepted by the first portlet, the Common Operation Picture (or COP), have it notify the host and have the host ask all the portlets involved to render the right-click menu. The COP portlet should pass enough information as part of the event so that the other porlets would be able to do something meaningful with it. Both options are valid the second is usually simpler to implement as you don't have to interfere with the UI framework to ensure the host will get the events.

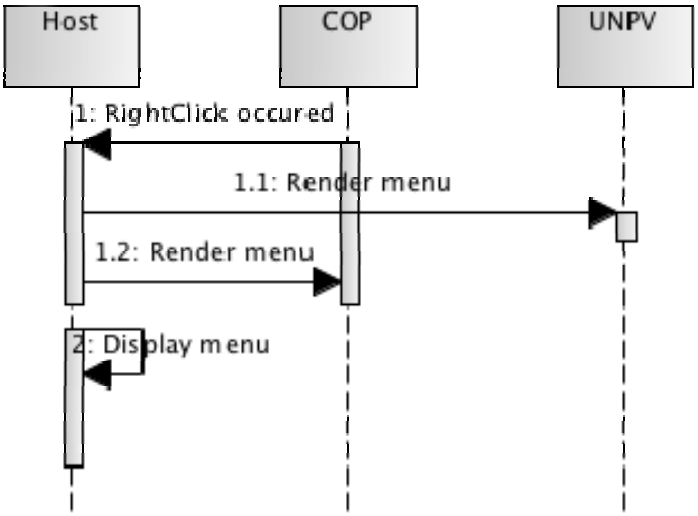


Figure 6.6: Sample event flow in a Composite Frontend. Events are intercepted by the user interface components of individuals portlets , The events are transferred to the host which dispatches them to registered protlets for handling. The host can then render the results for display

The composite frontend pattern is a service consumer pattern so the proxy will utilize the various service interaction patterns like saga, request/reply etc. (see chapter 5) and it can benefit from the various service composition patterns like Service Registry and Servicbus (see chapter 7)

You've probably noticed the use of the term portlets to describe the service agents and you might be wondering why the pattern is named Composite Frontend rather than Portal. The main reason is that the pattern can also be used with rich client implementations and not just web ones – let's explore that further in the technology mapping section

6.2.3 Technoogy Mapping

Normally, you won't be developing your own Composite Frontend container. Instead you'd use existing products that provide the framework and usually the tooling to help build the portlets. The obvious example for that are web portal frameworks. Modern enterprise web portals usually support anything from JSR 168/286 (Java Portlet specification) to WSRP (Web Services for Remote Portlet) to open web standards like RSS, plain REST services or standards like open social. There are a lot of products in this area both commercial like ibm WebSphere portal server and Microsoft Sharepoint and open source options like Jboss Gatein

and Lifray. Figure 6.7 shows the layout functionality of the UI host as it is implemented in Jboss Gatein

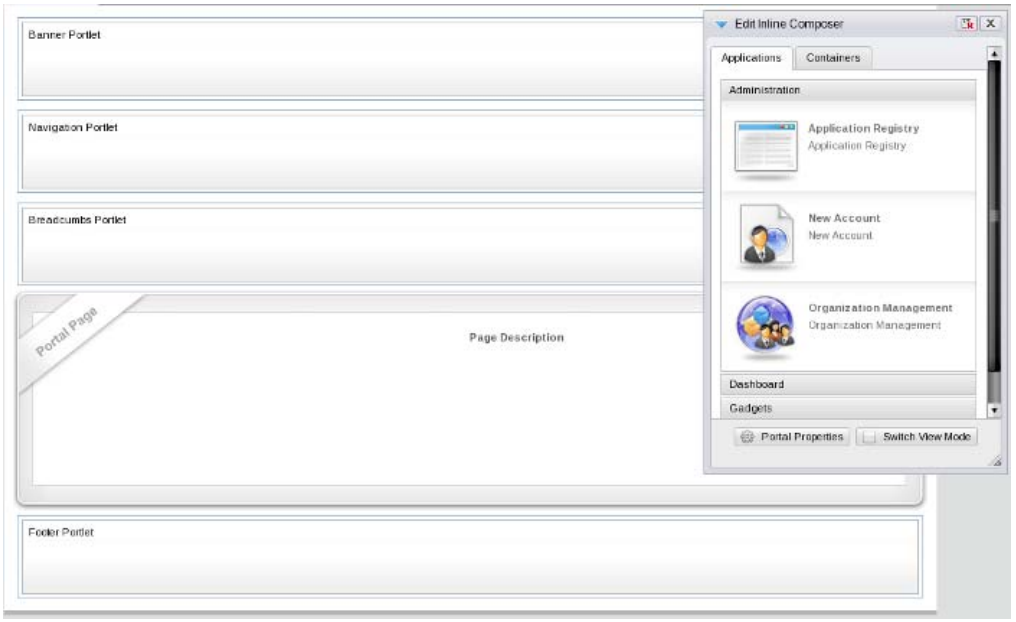


Figure 6.7: The layout capability of a Composite Frontend UI host as it is implemented in Jboss Gatein portal.

Web portals are not the only option for implementing composite frontends. You can also implement the concept for desktop ("rich client") applications. An example for that is the prism framework made by Microsoft's Pattern and Practices group Prism implements the Composite Frontend pattern for both Silverlight and WPF applications. Prism provides all the functionality of a user interface host and lets you write portlets that consume these capabilities. Code snippet 6.4 below demonstrate using an EventAggregator facility that allows inter-portlet communications (such as the one needed for the sample scenario in the map component example above):

Code listing 6.4 Sample use of Prism's EventAggregator

```
[Export(typeof(SampleView))]
  public partial class SampleView : UserControl
  {
      [ImportingConstructor]
      public SampleView([Import] IEventAggregator eventAggregator)
      {
          InitializeComponent();
```

```
            eventAggregator.
                GetEvent<CompositePresentationEvent<ItemSelectedEvent>>().
                                    Subscribe(OnItemSelectedReceived); #1
        }

        public void ItemSelectedReceived(ItemSelectedEvent item)
        {
                //do something with item...
            }
    }
```

#1 subscribing to an temSelectedEvent. Another portlet can call get on the EventAggregator to get a reference to the event and raise it without knowing if thare are any subscribers

in addition to web portal frameworks and desktop frameworks you can roll your own implementation of Composite Frontend. However, as mentioned above, it is usually better to choose one of the available options as it is quite an investment to get it right.

6.2.4 Quality Attributes

Before moving on to the next pattern lets examine some business drives (or scenarios) that can drive us to use the Composite Frontend pattern.

In essence, the main drives to Composite Frontend are flexibility to adding and changing services and the desire for an integrated user interface that feels as a whole Table 6.3 provides examples for both quality attributes:

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Usability	Operability	Under normal system use end user wants to achieve business tasks fluently. System should reuse entered data (like personal details) between different tasks
Flexibility	Changability	Under normal conditions, changing the billing process to support a new credit card clearance provider, should take less than one week

Table 6.3 composite frontend pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using composite frontend pattern.

Composite Frontend is probably the preferred way to provide an SOA user interface. However one problem we still have to solve with integrating UIs is UIs that are not SOA aware – for instance what happens when we have an existing UI that we want to expose to services? The next pattern will try to answer exactly that.

6.3 Client/Server/Service

It is always nice to work on “green-field” projects. Green-field project means fewer constraints. No existing systems that you need to work with/around. Most projects, however, are not like that. We do have systems in place and existing assets we need to integrate and work with. This is especially true for SOA projects which are usually large transition projects that happen gradually over time – after all no one will stop the enterprise while you get ready to ship.

The Composite Frontend pattern above takes a look at building a user interface for SOA in a manner that is akin to a green-field project, creating a new user interface, from scratch, to consume newly developed services.

As usual, lets start by presenting a scenario to get a better grasp on the problem.

6.3.1 The Problem

I worked on a project where the company just finished converting their user interface to a 3-tier solution, based on Microsoft Silverlight connected to an application backend. Our team on the other hand, was tasked with building new services as well as replacing existing business capabilities with new services that add additional functionality. To help complicate things, the chosen technology for the new system was Java and related technologies. Figure 6.8 shows a simplified illustration of the problem.

On the right side we see the current system that has components for single-sign-on (SSO), and some business logic to handle customers, orders and invoices. On the left side we see the services that are going to be developers with Ordering and Customers that are due to subsume and expand the current implementation as well as SalesRep service that introduce new business capabilities.

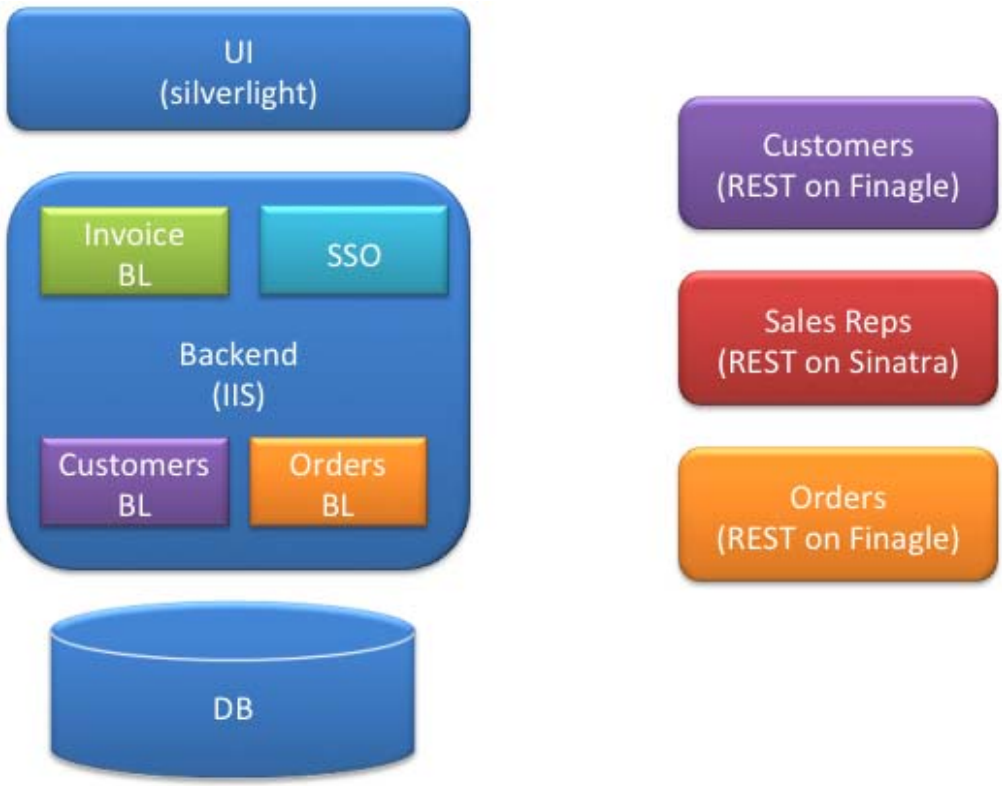


Figure 6.8 A sample for a 3-tier system that needs to integrate with new services. Some of the capabilities of the 3-tier system will remain intact (e.g. invoices), some will be migrated and expanded (like customers) and some would be added (like Sales Reps).

Our "dream" solution might be the Composite Frontend pattern (described above) where we have a portal-like user interface that will directly integrate with all the new services. This can be a possibility if the current architecture and technologies of the current user interface are compatible. For example for the project descried above, if the user interface was based on Prism and the backend services were based on ASP.Net then it would have been possible to stitch the new services to the existing system. Most of the time that's not the case and the questions we're left with is:

How do you connect an SOA to user interfaces where integration is problematic e.g. the client side is not SOA aware or it uses incompatible technologies

We've mentioned the possibility of not compromising on the user interface for the services and building the optimal service user interface, unfortunately this would result in a major rewrite and a long wait until all of that rewrite is done before the business users would get to use the new capabilities (long time to market). Not to mention that even in this simplistic example above, it is likely that not all the existing functionality is planned to move to SOA (or only planned to move in the long term) which can be another barrier for this kind of move.

Another option is to integrate the services within the existing user interface. *The main* problem with that is that it is very hard to maintain a cohesive and unified user experience when you are integrating two user interface concepts together. The secondary problem with this approach is the difficulty in integrating technologies due to tools or skillsets of the services and ui developers.

6.3.2 The Solution

We need to find a way to integrate the new functionality, begin the SOA transition and get a reasonable time-to-market. The answer in most cases is to:

Apply the Client/Server/Service pattern and use an intermediate server between the UI and the Services.

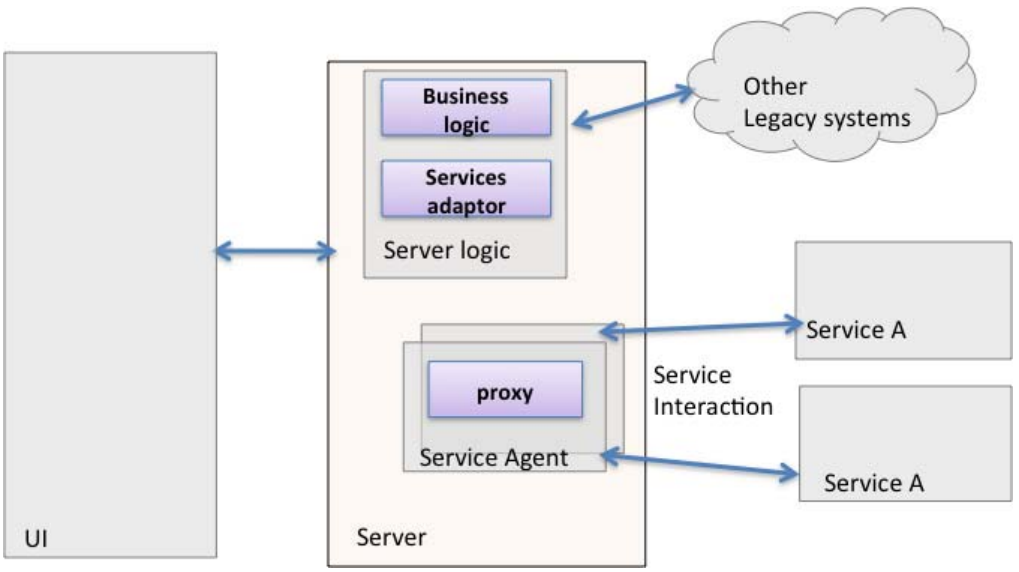


Figure 6.9 The client/server/service pattern. Integrate new services on the server side to minimize the impact on existing user interfaces and functionality.

The Client/Server/Service pattern is a simple pattern. It basically suggests integrating existing user interfaces with new services on the server side of the existing user interface. Essentially on the server you'd have a service agent, which is the proxy of each service. The service agent includes a proxy that is used for communicating with the services.

The existing service logic on the server will then be changed to integrate with the new service. The recommended way to do that is to change all the writers to write both to the existing implementation and the new one, then get all the readers to read from the new implementation and finally retire the old readers. This way you can always keep the application running and operational while you are making changes. "Wait a minute", you might say – "how is that different/better than integration at the user interface side?".

The main reason for that is integration and security usually the communications mechanism from client to server needs increased security vs. server-to-server integration (inside the firewall). Switching security contexts and maintain single-sign-on can be tricky cross technologies. Additionally, server integration has a wider selection of integration technologies than client-to-server. Lastly, it comes to smart-clients there are even more reasons such as increased complexity of maintaining uniform look and-feel when integrating different architectures.

Looking at integration with other patterns the Client/Server/Service works well with the various service integration patterns (see chapter 7) and utilizes the Service communications patterns (see chapter 5). It can also make use of the identity Provider pattern (chapter 4) for passing security context between the existing system and the new services

Let's take a look at our sample scenario and see how Client/Server/Service can be implemented.

6.3.3 Technology Mapping

Implementing the Client/Server/Service pattern does not require any specific technology, though as mentioned above employing integration patterns can help. In particular, tools like enterprise service busses (see Servicebus pattern in chapter 7) can help weave disperse technologies and architectures together. On the other hand, you can also integrate services and existing systems by building on simpler concepts like REST. Let's recap our sample scenario. As illustrated in Figure 6.10 we have a Silverlight service in a classic 3-tier setup and we need to integrate it with a few new services.

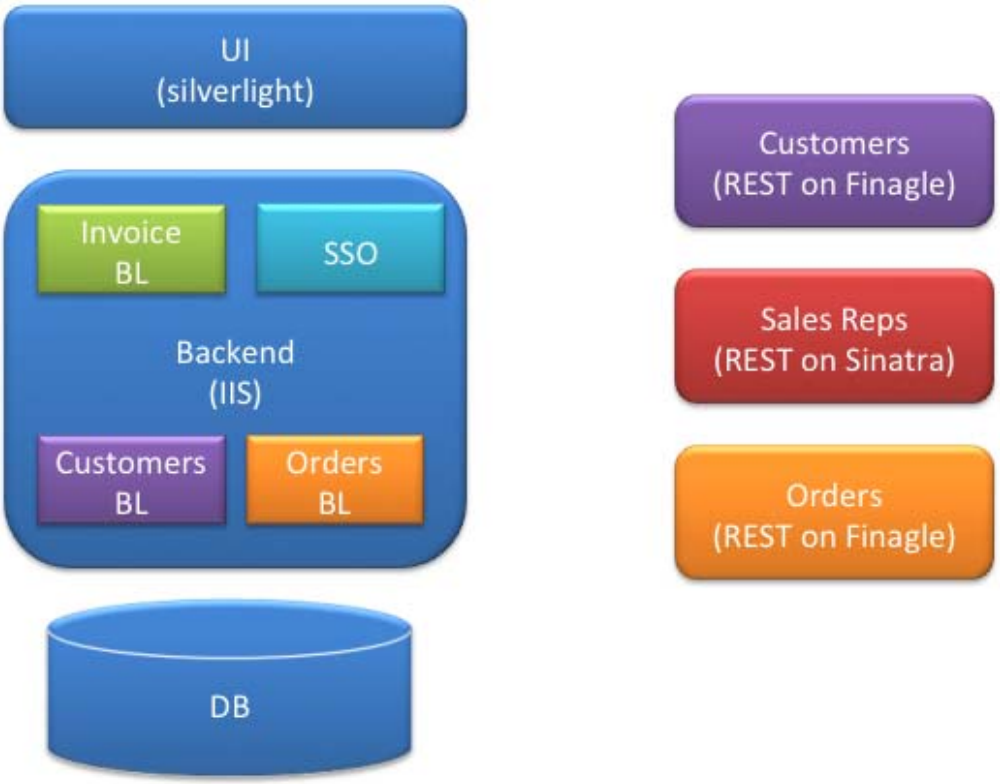


Figure 6.10 recap – we have a silverlight application build in a classic 3-tier model that needs to interact with new services.

The sample scenario mentions 3 different services: customers, orders and sales reps. Let's look at a few code excerpts of how we introduce the Sales reps service into the existing system.

The first place we need to change is the user interface itself. The listing below shows a short excerpt from the ViewModel of the sales rep page in the Silverlight user interface. We see that it has a WCF proxy that exposes "web-services" for the remote business logic. In the listing we see the call to retrieve the sales reps that are on the current shift.

Listing 6.5

```
public class SalesRepPageViewModel : INotifyPropertyChanged
{
    private ISalesReps salesRepService;

    private ObservableCollection<Employee> salesReps;
    public ObservableCollection<Employee> salesReps
```

```
        {
            get{return salesReps;}
            set
            {
                if (value != salesReps)
                {
                    salesReps = value;
                    RaisePropertyChanged("salesReps");
                }
            }
        }

        //Using dependency injection to tell us which implementation of the
    model we are using
        [Inject]
        public PageViewModel(ISalesReps proxy)
        {
            salesRepService = proxy;
            salesReps = salesRepService.GetCurrentShift();
        }
    }
```

The sales rep service on the other hand uses a mix of a RESTful interface for setting and getting state along with AMQP queues to push events out to subscribers. For example the current shift is an event that is pushed whenever a new shift starts. The listing below shows an excerpt of the Ruby code that handles subscription registration and allocates (binds) queues

Listing 6.6 : code excerpt with a Ruby backend that handles the salesrep service

```
    class Sub < Sinatra::Base
        def self.init
          @@registered_queues={}
          @@rabbit_wrapper=Bunny.new
          @@rabbit_wrapper.init
          …
        end
        put '/salesrep/subscribers/:name' do |n|
          if  not @@registered_queues.key?(n)
            @@registered_queues[n]=@@rabbit_wrapper.allocate_queue n
          end
          status 200
            # return hyperlinks to subscriptions and subscribers...
        end

        get '/salesrep/subscribers' do
          puts "Subscribers list"
          if not @@registered_queues.empty?
             @@registered_queues.each { |queue| puts queue }
          end
        end

        post '/salesrep/subscriptions/:name'  do |n|

          request.body.rewind
```

```
        values=request.body.read.split(",").each do |agent_id|
         @@rabbit_wrapper.subscribe_topic n, Topic+sales_rep_id+".#"
        end
        status 201
        #return ref to the subscription
      end
    end
```

In order to tie the Silverlight (C#) code with the AMQP messages received we have a proxy on the business logic server that looks like a regular WCF service from one side so that the silverlight client can interact with it and also uses REST and AMQP on the other hand to communicate with the service. The listing below shows an excerpt from the mediation code, written in C# that creates subscriptions for changes in Sales Reps

Listing 6.7 excerpt from the Sales Rep service proxy on the backend service

```
public static void Subscribe(string subscriberName, string s)
          {
              var addr = new Uri(HOST_URI, SUBSCRIPTIONS + subscriberName);

              var req = CreateHttpRequest(addr, new TimeSpan(0, 0, 0, 30),
  WebRequestMethods.Http.Post);
              AddBody(req,s);
              var response = CallApi(req);

          }
public static void AddSubscriber(String subscriberName)
          {
              var addr = new Uri(HOST_URI, SUBSCRIBER+ subscriberName);

              var req = CreateHttpRequest(addr, new TimeSpan(0, 0, 0, 30),
  WebRequestMethods.Http.Put);
              var response = CallApi(req);
          }
```

Naturally, there's a whole lot more code there, to provide the actual interface, mediate between the service and the ui and actually provide the business value, but the idea is that utilizing Client/Server/Service we can deliver the quality attributes we need and get a working system

6.3.4 Quality Attributes

The main drivers for the Client/Server/Service pattern are not technical – a pattern like Composite Frontend is technically superior way to provide services with a user interface.

Table 6.3 composite frontend pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using composite frontend pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Usability	Efficiency	When the user needs to learn new to use features, the

		experience should be streamlines to ensure minimal learning curve
Business drivers	Time-to-market	The time-to-market of the new changes should be X months

An additional reason for utilizing the Client/Server/Service, which is not directly related to quality attributes is specialization of the development teams. If we indeed have teams that are adept in different technologies we may want to minimize the interfaces between the teams and thus a centralized "access point" provided by Client/Server/Service can help achieve that.

> **NOTE**: it is important to remember that the Client/Server/Service pattern is usually a transient pattern. In these cases it is just a stepping stone while making the move to SOA from an existing system.

6.4 Summary

Chapter 6 coverd three patterns related to how service consumers can better integrate with services.

- Reservation deals with providing time-bound guarantees that allow consumers to work and coordinate with several services (while avoiding distributed transactions)
- Composite Frontend describes a pattern for integrating user interfaces with services in a way that keeps the SOA premise for agile integration and adaptability
- Client/Server/Service shows a way to deal with the transition period of moving from n-tier architectures to an SOA while avoiding large rewrites.

Naturally service consumers have a lot of other relevant patterns. For instance user interfaces have patterns like Model-View-Controller (and related like MVVM, MVP etc.) most of these patterns are not directly related to SOA. One notable pattern/concept which I recommend exploring is Command Query Responsibility Segregation t (see further reading)

The other patterns related both to consumers and services amongst themselves are described elsewhere in this book. For instance, the communication patterns described in chapter 5 are another type of pattern relevant both for service consumers, and services are the Service integration patterns that you can read about in the next chapter.

6.5 Further reading

This section provides links to resources (web or otherwise) for all the technologies discussed in this chapter.

Table 5.7 resources for further reading on topics covered in this chapter.

Topic	Resource name/link	Why
Composite Frontend	Open social	One of the technologies discussed
Composite Frontend	Prism http://compositewpf.codeplex.com/	A desktop implementation of composite frontend pattern
Composite Frontend	http://silk.codeplex.com/	Project silk – a web equiv. of prism.
Client/Server/Service	Silverlight Java interop http://www.infoq.com/articles/silverlight-java-interop	A sample integration mentioned in the pattern
summary	Cqrs http://martinfowler.com/bliki/CQRS.html	A complimentary approach for client-service communications

7

Service Integration Patterns

The previous chapter looked at how Service Consumers integrate with Services to achieve their goals. Chapter 7 continues this and takes a look at higher-level integration of services to achieve goals that are beyond the goals of a single service for instance the collaboration of several services to create a complete business process or creating a report on information from multiple services.

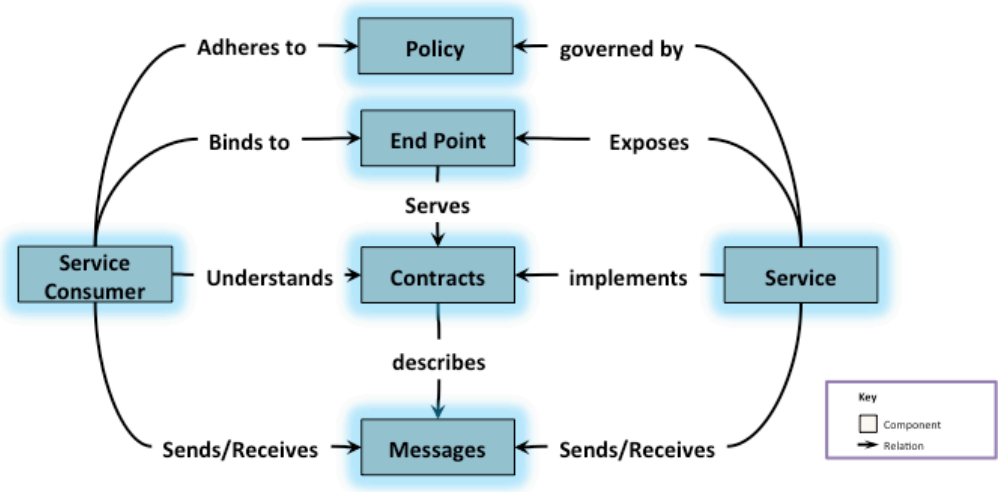


Figure 7.1 Chapter 7 talks about service integration – that is, connecting and making services work together to achieve business goals.

Table 7.1 summarizes the patterns discussed in this chapter and the challenges that they address

Table 7.1 list of patterns covered in chapter 7.

Pattern name	Problem addressed
Service bus	How do we make services interact in a decoupled manner over different protocols, dynamic configurations and routing?
Orchestration	How to make business processes agile and adoptable while using services based on request/reply or request/reaction interaction patterns
Aggregated Reporting	How do you get efficient business intelligence and summary reports spanning across the business when the data is scattered and isolated in autonomous services?

The first pattern we will look at is the ServiceBus pattern, which is a communication building block, that helps services integrate and collaborate together.

7.1 ServiceBus

Congratulations, you are starting a new enterprise and you think that it would be a good idea to model it using SOA. Since there aren't any legacy systems around, you choose <insert your favorite technology here> , a messaging technology to match (JMS/REST/WS etc.) and your are all set. You have got a heterogenic environment. Each of the services you develop can easily talk to the other services because they are all built using the same technology stack.

7.1.1 The Problem

Homogeneity, that's a reasonable assumption when you develop everything, right? Well, as it happens, sometime early in the 1990's Peter Deutsch and a few others drafted the "fallacies of distributed computing". These are 8 assumptions newcomers to distributed computing tend to make which prove wrong in the long run. One of these fallacies is "network is homogenous". Homogeneity will hold for a while, but then you'd have to integrate with a 3rd party vendor, your company will merge with another or you'd just have to integrate with an existing legacy system or maybe just the technology will go to v.next. Regardless of the reason and no matter what you've started with, sooner or later you'd find yourself in a situation similar to what's illustrated in Figure 7.2 below – A bunch of services using different technologies not all of which you control – and the task to integrate them all. The initial thought when this happens might be to just add an additional endpoint. Service 3 in the diagram, does just that – and has both a WS-* endpoint and a RESTful one so both service 2 and service 1 can interact with it. This still doesn't solve Service3's own problem of

consuming services from Service1 and Service2 because it still needs to support the two protocols. Not to mention what happens

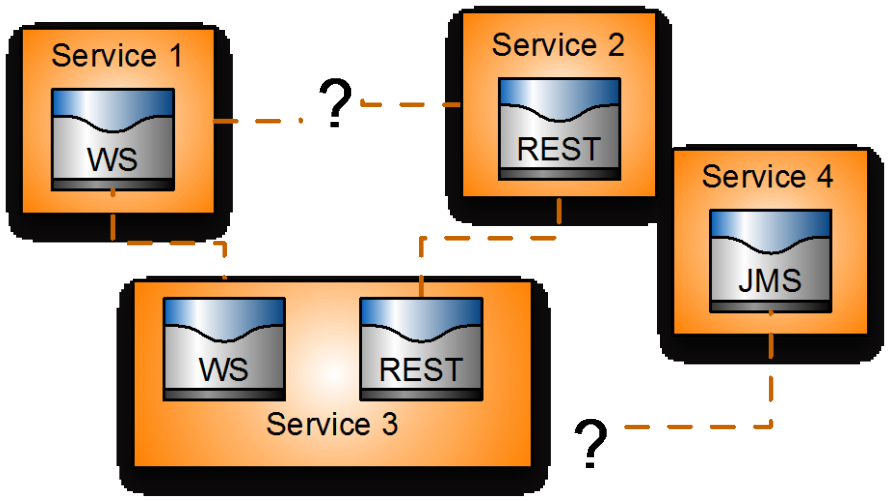


Figure 7.2 Tower of Babylon, service style – services of different types WS* web services, REST, messaging that need to be integrated somehow. Some services may have more than one endpoint like service 3 – but most won't. Even if you choose to have two endpoints for each service what will you do when the third technology appear?

Different communication protocols, as depicted in figure 7.2 above it just one problem we can encounter when trying to get services integrated, other examples can be bridging different security protocols, message transformation (like XML to JSON or handling of big decimals in various platforms and technologies).

Another related problem has to do with message routing, especially if/when were using the Inversion of communications pattern (see chapter 5). If we use subscriptions and messages, having each and every service manage these subscriptions is an overhead not unlike supporting multiple protocols in each service that we've mentioned above.

Essentially to solve all these types of problem we need to find a way to get different services interacting regardless of protocols, languages etc. We can state the general problem as follows:

How do we make services interact in a decoupled manner over different protocols, dynamic configurations and routing?

Multiple interfaces/endpoints for each service, as mentioned above, can be a good option when you want to make sure your service is usable from other services but it is not a good path to choose for integration. You can't control all the services (if we could we would be less likely to have a problem getting them all to speak the same protocol..) in the best case scenario you only solve half of the problem i.e. other services can communicate with your services but you still need to figure out and write integration code for services you want to consume. And you have to do that for each service – isn't that the point-to-point integration hell we wanted to escape from with SOA?

Ok, so we need some central piece of software to do the integration. One such option is ETL tools (extract, transform, load tools – see expanded discussion in "aggregated reporting" pattern in section 7.3) but they are batch oriented and SOA messaging needs to be real-time or near-real-time.

7.1.2 The Solution.

Basically we need the same ideas that ETL provides but applied to SOA – we need a Service Bus:

Implement the Service Bus pattern and use a unified messaging infrastructure, for message transformation, mediation, routing and invocation infrastructure

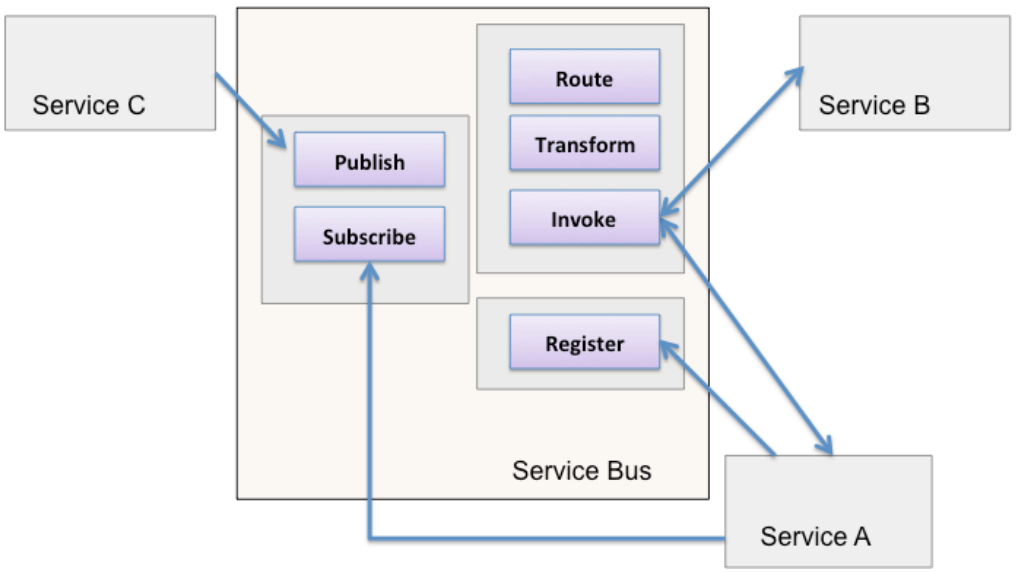


Figure 7.3 service bus pattern. Services can interact with each other using the bus as an intermediary. For instance Service A registers its endpoint with the Service bus and subscribes to messages such as ones published by service C. Now both new messages from Service C and requests from Service B can find their way to service A. either directly or by being routed and transformed before the actual invocation of service A.

The crux of the service bus is abstracting away the communications between services. To achieve that, the service bus pattern is basically a combination of several enterprise integration design patterns such as

- message bus – to connect the different services,
- message router – to know what message to send where
- channel adaptor to convert formats and protocols.

The Service Bus pattern is composed of 4 main roles. The first role is, we have service registration. The bus needs to know where to find services so that it can invoke them. It also needs to provide a facility for services to configure and expose additional endpoints that other services can consume.

The second role has to do with message handling. The service bus provides capabilities to invoke a registered services using the endpoint they've defined on the bus. The bus also needs to route the message to the registered service. Lastly, the service bus sometimes needs to perform protocol and/or message transformations to make sure that the targeted service can actually handle the message.

The third role is to facilitate publish/subscribe. For one, the bus provides subscription services (which can be thought as a type of registration) . Then when services publish messages, the service bus can use routing and transformations and invocation to call the subscribed services.

An architectural diagram of a solution that uses a bus will usually look something like Figure 7.4 below. A few services with a central "thingy" connecting them all.

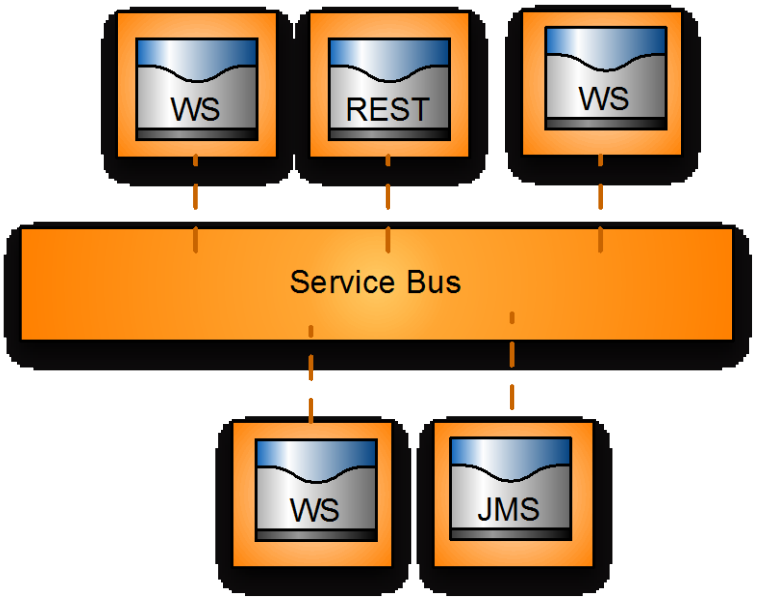


Figure 7.4: Typical representation of a Service Bus - a single entity which all the services connecting to it

However, in reality there are three deployment options for the service bus pattern as illustrated in figure 7.5: Hub and spoke, Bus (peer-to-peer) and Federated (mix of the other two)

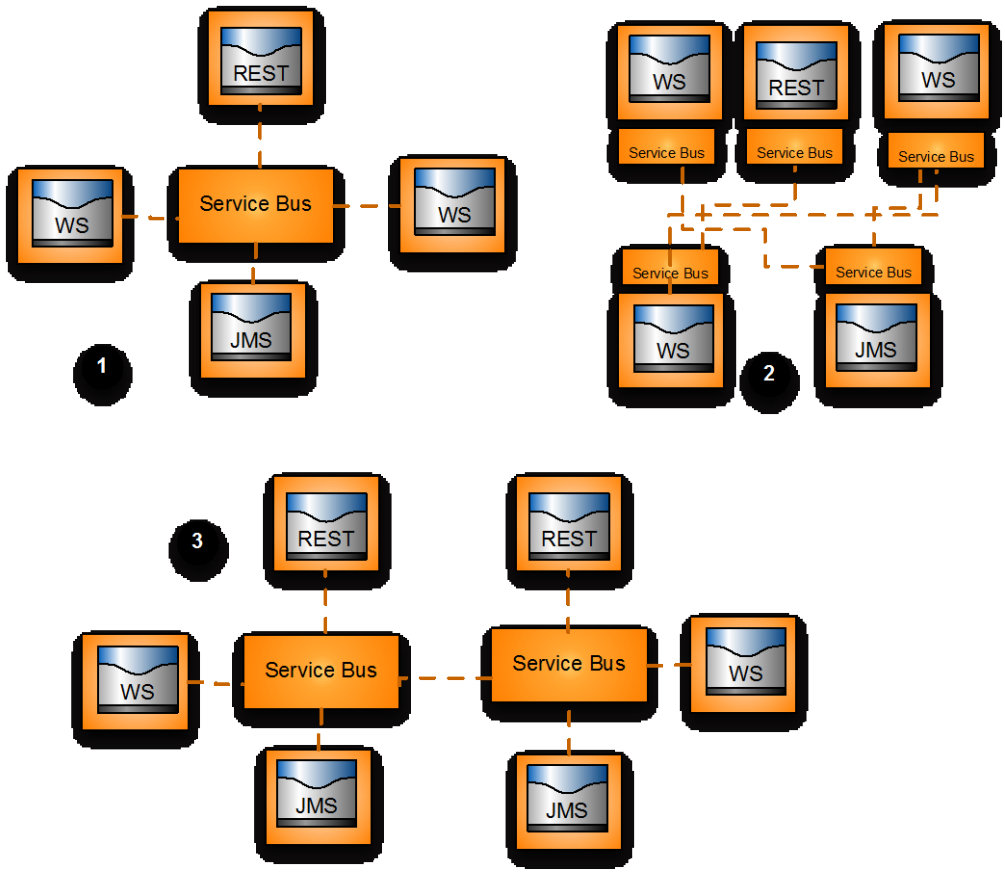


Figure 7.5 Three possible deployment topologies for the service bus pattern. 1. A hub and spoke setup with a central service. 2. A bus where each service has a service bus (agent) nearby and the service buses connect amongst themselves in a peer-to-peer manner 3. A federated setup. Each service bus is a central service for a few services but the busses also coordinate so that services can also communicate with services connected to remote busses.

The most common set up is Hub and spoke since it is easier to deploy and maintain. However the other two options do have some merits that you should consider. Naturally, each option also has some drawbacks, table 7.2 below, contrasts the three deployment options

Table 7.2 Service bus deployment architectures comparison.

Deployment	What is it?	Pros	Cons
Hub and spoke (option 1 in figure 7.5)	A centralized server (or two for availability) which all the services connect to.	Easy management Easy to debug	The server is a bottleneck (all traffic goes through it. Limited scalability (only scale up)
Bus (option 2 in figure 7.5)	Each service has a bus or a bus agent instance. Services communicate with their local service bus and the service busses connect with each other so they seam like a single network .	Scalability Flexibility High-availability (with store and forward)	Complex topology Relatively hard to configure and debug
Federated (option 3 in figure 7.5)	Small hubs that are interconnected For example you can have a hub-and-spoke on the departmental level and connect them to a federated solution across the enterprise	Scalable Simpler than the bus option.	More complex than hub and spoke

The service bus pattern adds a level of indirection so it has an effect on overall latency of operations. On the other hand it provides a lot of benefits in terms of flexibility and decoupling. You can choose not to use a service bus if you have a small system with just a few services – in most cases, though, you'd want a service bus of some sort in your SOA implementation.

Unlike most of the patterns in the book (but like most of the patterns in this chapter), you're most likely to implement the service bus pattern by choosing an off-the-shelf product and integrating it in your solution – rather than implement one yourself (though that sometimes happens see for example the case study in chapter 9). Let's take a look at some of the options available today.

7.1.3 Technology Mapping

Service bus implementations come in three main flavors: message busses, pure service busses and ESBs. Table 7.3 below provides a brief explanation of these flavors.

Table 7.3 service bus implementation flavors

Implementation type	Details	Sample products
Message bus	message oriented middleware or solutions built atop message oriented middleware can be used as service buses when your services use messaging (see Inversion of communications pattern in chapter 5)	Java– ActiveMQ, MQSeries .NET– masstransit Other – RabbitMQ, 0MQ
Service bus	In contrast to message busses, service busses support SOA concepts like contracts. While they can support publish/subscribe they also support request/reaction and request/reply patterns	.Java – apache cxf + Apache camel .Net – nServicebus, Microsoft Azure service bus
ESB	Service busses which also support additional patterns/capabilities packaged as a single product (see callout below for more details)	Java – mule, fuseESB, websphere ESB, .NET – Neuron ESB,

Looking back at SOA projects I've reviewed I'd say the most common implementations of the service bus patterns use ESBs. The most likely reasons for that are that most vendors have ESBs and that ESBs offer more features.

The most common deployment model I've seen for the service bus pattern is a hub-and-spoke. The reasons for that are that this is the most common deployment model for ESBs as well as for to licensing prices (ESBs are usually priced per server so if you'd have a lot of them it can get costly).

Nevertheless, when you evaluate service bus implementation I'd recommend not to dismiss pure service busses as having service instances running on each server goes a long way towards better performance and flexibility.

ENTERPRISE SERVICE BUS

Enterprise Service Buses (ESB) are products that cover a lot of service infrastructure aspects in addition to implementing the service bus pattern. In addition to service mediation and routing, ESBs will usually add capabilities like orchestration (see orchestration –the next pattern), provide management capabilities (see Service Monitor pattern in chapter 4), help deliver some reliability features (such as Virtual Endpoint mentioned in Chapter 3) etc.

Additionally ESBs usually come with a lot of connectors for easy integration with 3rd party systems and protocols.

As mentioned above, you're more likely to buy/get a service bus that implement one. However you still need to configure it. Sometimes that would be done using visual designers, sometimes in xml and sometimes in code.

For instance code excerpt 7.1 below shows using apache camel Scala DSL (Domain Specific Language) to configure a simple route for routing messages according to the tenant Id (assuming an earlier route authenticated the request so that the tenant id is correct)

Listing 7.1 : Routing with Apache Camel

```
package com.rgoarchitects.camelDemo.routes
import org.apache.camel.scala.dsl.builder.RouteBuilder

class SlaRoute extends RouteBuilder {
  when (req=>RouteValidator.checkSlaLevel(req.in("TenantId"))==High)  -->
"http://betterServiceUri"
  otherwise -->   "http://waitInLineUri"
}
```

For another example we can see the code excerpt 7.2 below. Excerpt 7.2 shows the XML configuration for declaring a Jersey (REST -JAX-RS JSR-311 implementation) endpoint in mule ESB. This sample configuration file uses the Jersey connector of Mule (http://www.mulesoft.org/jersey). The interesting lines here, start with the service tag where we configure an inbound endpoint and tell mule to call our class (com.rgoarchitects.sample.Categories). The result of this configuration is that going to http://localhost:8991/ will go to a categories REST API defined in the class. Note that , once we have the endpoint set up we'd probably wrap that with a route that will do authentication and authorization and expose it to the outside world.

Listing 7.2 declaring a REST endpoint in mule ESB

```
<?xml version="1.0" encoding="UTF-8"?>
   <mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         xmlns:jersey="http://www.mulesource.org/schema/mule/jersey/2.2"
xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"

        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.mulesource.org/schema/mule/core/2.2
        http://www.mulesource.org/schema/mule/core/2.2/mule.xsd
        http://www.mulesource.org/schema/mule/jersey/2.2
        http://www.mulesource.org/schema/mule/jersey/2.2/mule-jersey.xsd
        http://www.mulesource.org/schema/mule/vm/2.2
        http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd">

      <model name="CategoriesResource">
           <service name="categoriesResource">
           <inbound>
```

```
                <inbound-endpoint address="jersey:http://localhost:8991/"
synchronous="true"/>
            </inbound>
            <component class="com.rgoarchitects.sample.Categories"/>
        </service>
    </model>
  </mule>
```

Lastly figure 7.6 below shows a screenshot of nServicebus (a .NET service bus implementation) modeling tools for Visual Studio which can be used to design publishers, subscribers and messages for service to service interactions

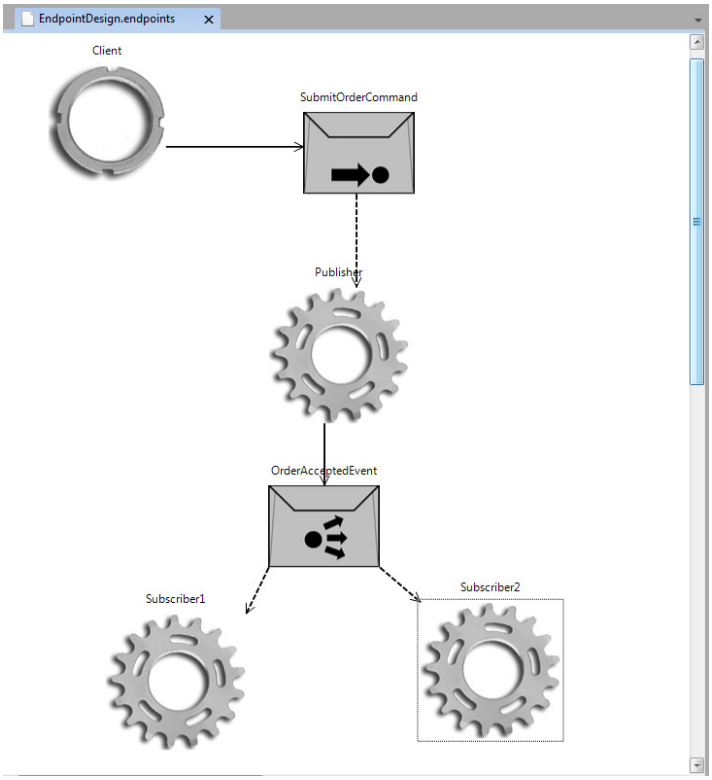


Figure 7.6 : Designing message flows, subscribers and publishers with nServiceBus tools for visual studio. The designer shows 2 messages – a submit order command and an order accepted event as well as a publisher service with two subscribing services

To summarize there are many service bus pattern implementations out there, and they come in all shapes and sizes. Depending on your technology stack and needs you can most likely find a solution that will work for you.

7.1.4 Quality Attributes

Before we move to the next pattern lets have another look at some of the reasons for opting for a service bus pattern. As usual we'll do that by looking at the quality attribute perspective.

In essence, the main drive to use a service bus is the need for loose coupling of service interactions. The service bus pattern provides flexibility, openness and contributes towards adaptability of SOA implementation. Table 7.4 below provides a few quality attributes scenarios to demonstrate these quality attributes.

Table 7.4 Service bus pattern quality attributes scenarios. These are the architectural scenarios that can prompt us to use the Service bus pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Availability	Effort to change - deployment	Under normal conditions, adding a server for scaling purposes should take no longer than 4 hours (including installation, configuration, etc.)
Changeability	replacing component (vendors)	During development replacing the credit card processing gateway should take one week or less.
Flexibility	Interfaces	During development, adding REST api to the system should be supported.
Interoperability	Integration	During development system x should be connected to system y, z, w

Another integration pattern that is geared towards flexibility is the orchestration pattern. Lets explore that next.

7.2 Orchestration

ServiceBus, the pattern presented in the previous section enables services to communicate in a decoupled manner. That's a good start – we've lowered the technical barriers to get services to talk to each other. The next topic on our plate are the business processes.

What's a business process? Well, services partition the enterprise capabilities to functional areas however for the business to accomplish anything meaningful we need services to work

together. Even a simple shopping cart scenario needs information from the customer, orders, invoices , inventory etc. A sequence of related messages between services to achieve a business goal is a business process.

The orchestration pattern provides a way to build business processes in a flexible way, bur first thing first, let's see what's the problem what makes us want something like this.

7.2.1 The Problem

On one project we worked on an e-commerce site for produce (fresh vegetable, fruit, dairy products etc.), which, once ordered, was picked up from participating farmers and grocers. Figure 7.7 below shows the base business flow once a shopping cart is submitted. The users fills a cart in multiple shops, submits the cart which creates an order. The order is then billed and lastly a delivery guy goes from grocer to grocer to pick up the goods and fulfills the order (reporting back to the system on the item that were delivered)

Figure 7.7 baseline e-commerce flow – the user fills a shopping cart and decides to order. He is then billed and then the order is sent to fulfillment.

This looks great, cool, simple and clean so lets say we develop that. We now have a cart service it calls the orders service, which does it thing and then calls billing etc.

If we'd do that in real life we'd soon discover that the process is wrong. True, when you order a carton of milk the above-mentioned process works. However, we're dealing with produce here so, for instance, when you order a kilo of tomatoes you might only actually get 0.96 kilo or 1.051 kilos. Also we're dealing with small businesses here so they might actually be out of a certain product by the time of pickup. So basically we need a new process where after registering the order we secure the order amount with the credit card company and during fulfillment we update the order and set final billing. So again we go back to all the services and update the call paths so that each will know about the other calling path

Now what will happen when we will enter another market and find out the fulfillment works in yet another way. Not to mention about the whole sales process where we also want to add promotions, coupons and whatnot.

Business processes are bound to change, either because we gain better understating of the business or because business requirement change (say a new competitor in the market) We can't go on hand wiring services to other services every time that happens - we need a way to make the business processes more flexible.

How to make business processes agile and adoptable while using services based on request/reply or request/reaction interaction patterns?

Obviously hard coding the interaction pathways as described above will not get us very far. The sample above was just one business process a solution would usually have quite a few of those and they'd be changing just as much. Hard coding is wrong as we have to

- Create new version of a service just to change the flow
- Business process is scattered about and hard to isolate
- Hard to change when the need arises

The previous section (7.1) introduced the servicebus pattern, which among other things provides message routing maybe we can use the flexibility it introduces to help solve our problem here. Well, routing at the servicebus level will help with most of the problem, namely, externalizing the routing logic and making it easy to change. In fact, for many cases, esp. when the business processes are simple, it can be enough (see also discussion on choreography below), however it will still limits the visibility of the complete business process making it hard to grasp the complete process esp. when the business is not trivial and/or there's some back and forth between services. Not to mention situations where the complete process also requires human workflow to complete properly.

To get all the properties we want we need something different we need Orchestration.

7.2.2 The Solution

Add an orchestration engine to externalize business processes from the services and allow controlling and changing them dynamically and provide governance on system behavior.

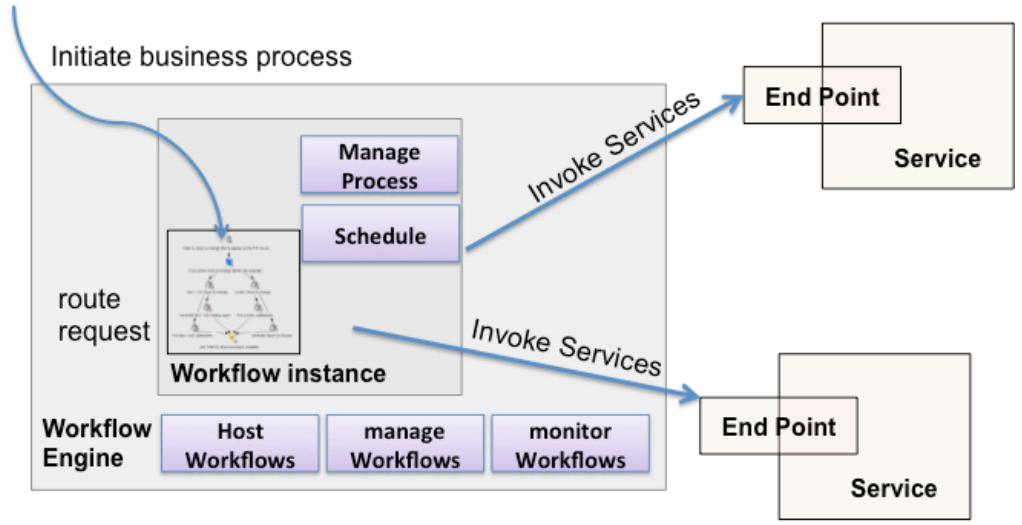


Figure 7.8 Orchestration Pattern – an external workflow engine activate a sequence (simple or compound) of services to provide a complete business service.

The Orchestration pattern is a relatively simple pattern – in essence it is about adding a workflow engine that is external to the services. Modeling the different business processes as flows of service interactions and letting the engine execute them to carry out the process, monitor and manage it.

The main component is the workflow engine. As mentioned above, it manages workflows, i.e. provides the means (usually visual) to define, edit and delete workflows. The workflow engine also host workflow instances and monitor their progress.

A process that runs is instantiated as a workflow instance which can support scheduling and managing the process itself e.g. forking (sending few requests in parallel), joining (waiting for replies or reactions from multiple services), handling failures etc. A workflow can be a short-lived process but in most cases it will be a longer running process.

We've already covered a pattern that talks about long running processes – the Saga pattern (see chapter 5). In a sense orchestration is a particular implementation of the Saga pattern where the coordinator is external to all the participants. The Saga pattern is more generic as it can also be implemented without a central knowledge of what should be done when to complete the business process. This has the advantage of having services that are more autonomous and emergent and flexible processes but the cost for that is lack of a clarity on what constitutes a business process and with that the ability to monitor and understand their current state.

Note that most workflow engine implementation do not hold the workflow instance live through its running but rather save it state between calls and retrieves it when a new message arrives (a process called dehydration and hydration).

What about event driven SOA?

The problem statement for the Orchestration pattern specifically talks about request/reply and request/reaction communication patterns – which begs the question what about pub/sub or event based communications (Inversion of communications pattern – see chapter 5). Orchestration is not a good fit in this situation

Well, if orchestration is a metaphor for a conductor telling each service what to do, events lends themselves to another arts related metaphor – choreography where each service plays its part; independently publishing events that occur within it and subscribing to events it needs to perform its role. The resulting "dances" are the different business processes of the organization.

Choreography is not described as a pattern in the book as is it is more an emergent property of using events than a deliberate pattern. Choreography provides even greater flexibility than orchestration does and it allows for emergent business processes and behaviors not planned in advance.

On the down side, it lacks the explicitness of business processes that Orchestration provides – to compensate for that and ensure the system will be correct services should be developed as autonomous as possible (see for example Active Service in chapter 2) to keep the problem each solves as localized as possible. Also it is recommended to externalize the events into an event catalog to allow both reuse and system-wide governance (see further reading).

Notations like BPMN2.0 (Business Process Model & Notation) support the design of choreographies in addition to workflows.

Figure 7.9 below shows the revised flow of order handling in the produce e-commerce solution mentioned in the problem description of the Orchestration pattern. As mentioned above, the actual process needs more coordination between the different services than the original flow shows. You can study the diagram for the details of the flow, however, the more important point here is that it is modeled as a workflow with several decision points that alter the process e.g. point 1 in the diagram where depending on thour ability to secure the funds for the order we can abort the whole ordering process. Changing the workflow will change the business process and, depending on the capabilities of the services involved, may not require any code changes. For instance point 2 in the diagram where a timeout on delivery cause the whole process to be aborted is a new requirement (vs. the description at the beginning of this section)

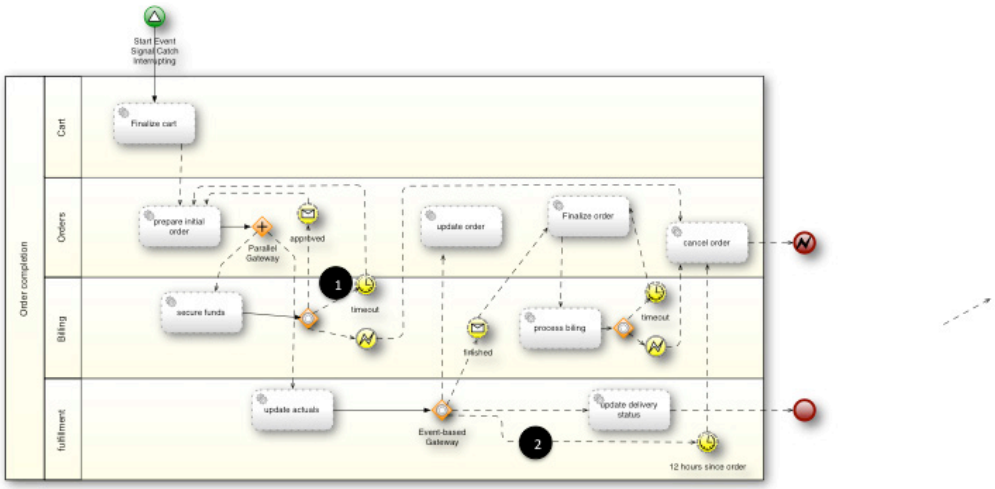


Figure 7.9 updated flow for processing produce orders. The flow is much more complex than the initial naïve version for instance when initializing the order we need to secure the maximum order value with an external credit card processing company and only when the order is finalized and we know exact amounts calculate and bill the actual value.

Used right orchestration can add a lot of flexibility to an SOA as well as keeps us from loosing site of the forest for the trees – the many services we may have in our system. The main risk we run when using Orchestration is over-use which can result in SOA anti-patterns like nano-services (see chapter 8). To help avoid this problem you can partition the workflows between the external flow (Orchestration) and internal flows (Workflodize pattern in chapter 3)

Orchestration also works well with the service bus pattern mentioned in the previous section. Like servicebus, you're most likely use orchestration by choosing an off-the-shelf implementations rather than implementing one from scratch.

7.2.3 Technology Mapping

Orchestration is implemented by two classes of tools Enterprise Service Bus engines that also provide some orchestration capabilities as business process management (BPM) systems that are built for handling orchestration and workflows. Choosing one type of tools over the other depends on the complexity of your processes, the needed performance etc. Table 7.5 below provides some general guidelines for each class of tools (specific products can be more versatile than the generalization below)

Table 7.5 Enterprise service buses vs. business process management tools as orchestration engines

	ESB	BPM
Main purpose	Integration and virtualization of services	Running and monitoring business processes
workflow	Basic workflows	Extensive support including loops, rules etc.
Performance	Built for high message flows	Built for complex processes
Human workflow	Not supported	Supported by some implementation
Saga support (long running interactions)	Suitable for supporting sagas in event based system by providing store and forward services	Suitable for supporting sagas by keeping track and following the state of long running interactions

It is important to note that if you can also combine both product types by having the ESB invoke processes that are managed by the BPM produce as well as having the ESB virtualize the end-points of the services used in the PBM process.

Another option is to use more basic workflow engines and build your own service orchestration on top – in most cases that's not the best option since it wastes a lot of effort especially since there are even open source options like jBPM available fore free.

There are two main notations used by various workflow or business process management tools "Business Process Execution Language" (BPEL) and "Business Process Model & Notation" (BPMN). A sample for a BPMN diagram was shown above in figure 7.9. BPEL is a more technical/developer oriented approach for describing interactions. Figure 7.10 below shows a simple BPEL process on a commercial designer (Oracle JDeveloper).

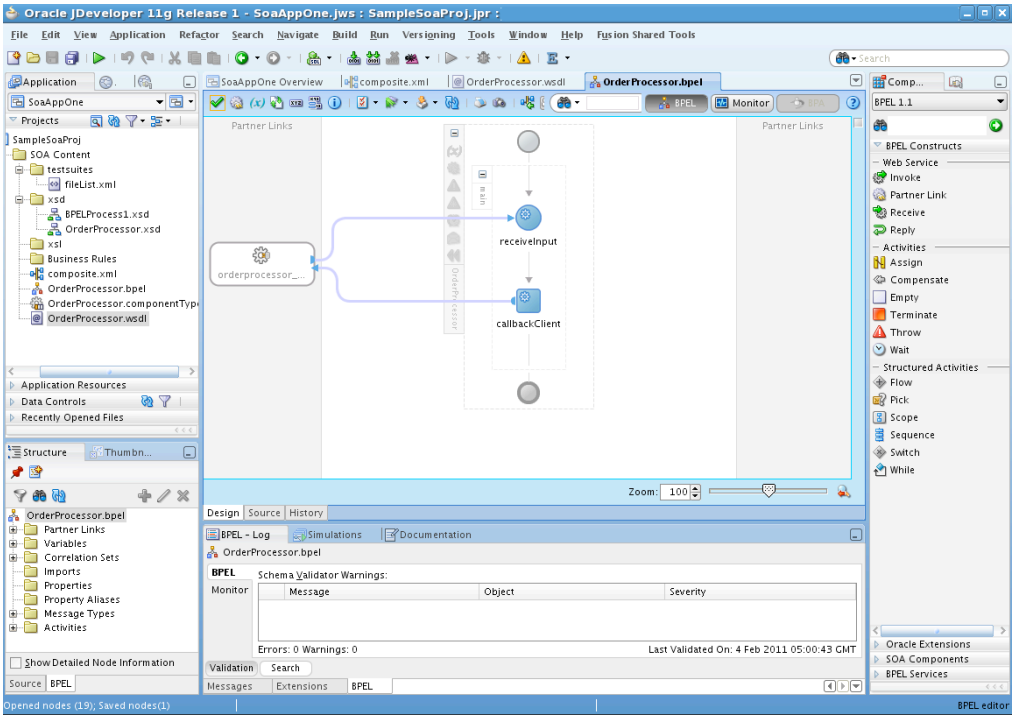


Figure 7.10 A sample BPEL diagram. Notice the detail level even on a simple flow vs. the high-level view of a BPMN diagram such as figure 7.9.

Both BPMN and BPEL are common notations. Some tools use one or the other and some tools support both. Table 7.6 below provides a short comparison between the two formats.

	BPEL 2.0	**BPMN2**
Notation characteristics	Developer oriented	Business people oriented – more abstract
Strengths	Low level concepts like compensation, fault handling, etc. Built for integration with ws-* standards	Easy to model complex interactions (higher level of abstraction), readability
Human workflow	Not supported	Supported
Standard	WS-BPEL by OASIS	Visual notation defined by OMG

REST Support	No – requires WSDL1.1 contracts	possible

When using tools that support both notations they can be combined so that BPMN is used to describe the process at high-level by business analysts and they are then expanded using BPEL by more technical people.

Regardless of the specifics of the technology used all orchestration implementation externalize the process from the services and provide flexibility to a SOA. There are few other quality attributes that orchestration promote and we'll examine them in the next section.

7.2.4 Quality Attributes

The main motivation for opting for the orchestration pattern is flexibility and the ability of the business to respond fast to changing business needs both a the macro level and at the practical technical level.

However, flexibility is not the only quality attribute promoted by the orchestration pattern such as increased run-time governance (by monitoring flows in progress), as well as increasing the chances to (re)using services in multiple processes. Table 7.7 below provides a few quality attributes scenarios to demonstrate these quality attributes.

Table 7.7 Service bus pattern quality attributes scenarios. These are the architectural scenarios that can prompt us to use the Service bus pattern.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Manageability	Understanding system's health	Under error conditions, an administrator will be able to understand problem and performance bottlenecks of different business flows
Changeability	replacing component (vendors)	During development replacing the credit card processing gateway should take one week or less.
Flexibility	Business flows	During development and operations, adding timeouts to all ordering processes will take less than 1 week.
Flexibility	composability	During development a developer will be able to find and reuse services in multiple

		business processes

The next pattern also deals with integration – it takes a look at how to get an integrated view of the data needed for reporting when SOA encourages each service to hold its own data internall.

7.3 Aggregated Reporting

Getting an SOA right is hard, not so much because of the technical problems (we know how to deal with those, right?), but rather since it is very hard to understand the business and figure out how to effectively partition it into services. Let's assume we, somehow, managed that and we have our business logic neatly divided into services. We then develop our business logic and business processes and all is almost done. All that is left is to produce a few reports… Well, maybe more than a few, perhaps dozens and dozens of reports. Assuming we did a good job partitioning our business into services, many of these reports will fall within the boundaries of our services. However, at least some of the reports will require data from several services.

7.3.1 The Problem

Let's try to visualize the problem. On a project I worked on we've built an analytics platform for call centers. The real-life system had a lot of services but for illustrating the problem we'll examine just 5 of them (see Figure 7.11 below)

Figure 7.11 : Services in an call center system. Customers makes orders and then call a call center to complain/solve problems. The customer's interactions with the call center representatives are recorders and analyzed.

- Voice interactions – all the past/current calls a person made to the call center. The service encapsulate both meta-data
- Customer – everything we want to know about the customer. Most of it is data imported from other systems like the CRM system etc. the customer service also does identity resolution (e.g. given a cell number find the user ID)
- Reps – The call center representatives. Data comes here from the operational system that the reps use when they interact with customers.
- *Orders* – Customer's orders.
- Classifications – classify calls according to business driven criteria e.g. calls by VIP customers that cancelled their service. Categories works in real-time (on incoming calls).It is task driven -i.e. any knowledge it has on customers or interactions is transient- it only stores the category definitions.

Now management wants to understand if there are any correlations between reps performance and loss of business by customers in general and VIP customers specifically.

All the information we need is in the system. The interactions contains all their classifications as well as the customer and rep ids – we can find which representatives handled which customers from there. Customers has the information on which customers are VIP customers and which aren't plus we can access orders to find out the business generated by each customer

Now all that's left to do is build the SQL query that takes all this data and produces the desired report. Uh uh, not so fast, There are 2 problems with this approach.

- One is the assumptions that all services use an RDBMS as a persistent storage. Few years ago that might have been a reasonable assumption, but today with the rise of NoSQL databases that may not be the case.
- The other, more important, problem with the "SQL to solve the report" approach is that it introduce a lot of coupling between the different services. Using a single SQL query to solve the problem, we need to know and understand the internal structures of each service. We constructed services with API level integration to escape this very problem.

The question is then, how do we generate reports in a way that does not violate SOA principles on the one hand produces the reports efficiently on the other hand:

How do you get efficient business intelligence and summary reports spanning across the business when the data is scattered and isolated in autonomous services?

One possible solution would be to create the report at the consuming end (e.g. the user interface). The consumer would call each service to get its part of the data and then perform all the grouping, crosscuts etc. This solution is rarely a good idea. It puts the burden of understanding the data and optimizing the query on the shoulders of each report consumer.

If we consider the example above, it is probably obvious that we need to go to customers first to get which of them are VIPs and only then get their total orders but should how do we connect that data to agent's performance ? which will yield a smaller set ? And that's just a single report…

Another option is the one we've already mentioned above- to go straight to the data, i.e. create an SQL query that will go into all the services' databases, join and get all the relevant bits. We've also seen why that's not a good idea either

Maybe the answer is "Aggregation Services" . This a notion that appeared in the early days of SOA and the idea is that when the granularity is such that the view of a entity is spread over multiple services (i.e. the granularity was wrong), we create a single service that creates a holistic view of that entity. The same idea can be applied towards creating an aggregated entity for the purpose of each report type – and we can copy over some data form all the relevant services (so we won't have the problem in mentioned in the first option). Well, Entity Aggregation was a bad idea for its original purpose and it isn't a great idea here as well. – for instance, who's the master of the data? Does each aggregate have its own copy of the data? Is the data federated from each service? What do we do when data changes? And if we can make it work – how many of these will we need to properly provide reporting capabilities?

Well the answer is that we can have one aggregated and to make it work it should follow some specific guidelines - I call this the "aggregated reporting service" :

7.3.2 The Solution

Create an Aggregated Reporting Service that gathers immutable copies of data from multiple services for reporting purposes

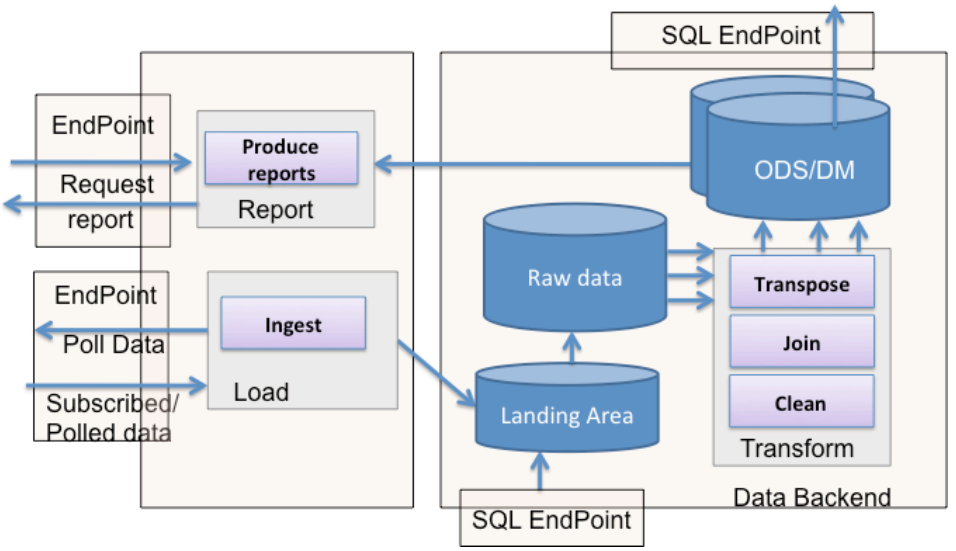


Figure 7.12 Aggregated Reporting pattern

Before we delve into the differences between aggregated reporting and "entity aggregation" and why aggregated reporting is a good idea, let's first understand exactly what this pattern is. Unsurprisingly, the aggregated reporting is about aggregating data from the services and then providing reporting facilities above the data to make it useful.

The pattern is made of two main components a Service and a Data backend.

The service component is the "SOA" endpoint of the aggregated reporting pattern. Where by "SOA" I mean it utilizes standard SOA technologies like web services or messaging. The service exposes two types of endpoints – The first type is an output endpoint which provides reports/queries that other services and service consumers can use. The second type of endpoint is an ingestion one. The ingestion endpoint allows collecting data from other service – either by subscribing to their events or by allowing other services the means to push data.

The second component, the "Data Backend" in the diagram above, is the core component of the aggregated reporting pattern. The component has 3 data stores. A landing area where some data from external interfaces is temporarily stored – this is done mainly for security purposes and to isolate the SQL ingestion endpoint from the raw data (we'll cover the endpoint a few paragraph below). Another data store is the raw data store. This can either be a temporary storage to coordinate data that arrives asynchronously or a long term data store that can be the basis for advanced analytics (answering questions we don't know yet we need to ask). The last data store is the reporting data store where data is kept in a reporting friendly structure – most likely an RDBMS).

The main active functionality of the data backend component is the Transformation service (which you can think of as an implementation of Active Service – see chapter 2). The transformation service responsibility is to rearrange all the incoming data in a way that would be useful for reporting over it. An aggregated reporting implementation would have extensive transformation components that will sift through the raw data, clean it, aggregate it and build useful representations of it

Last but not least, are the two endpoints of data backend -an ingestion endpoint for importing data and a reporting endpoint the allows querying data. These endpoints are unique as they use SQL and not technologies usually associated with service orientation. The main reason for that is that standard tools for both importing data as well as for business intelligence and reporting over data have been built on top of SQL for decades and forcing their hand to other technologies is usually not practical (in terms of effort vs. benefit). We still have to treat these endpoints as bona fide SOA endpoints though - isolate them from internal data structures, provide contracts etc. (we'll expand on this later)

There are a lot of components that play together here so let's take an additional look at the pattern from another perspective. Figure 7.13 below illustrates how data can flow into the aggregated reporting implementation.

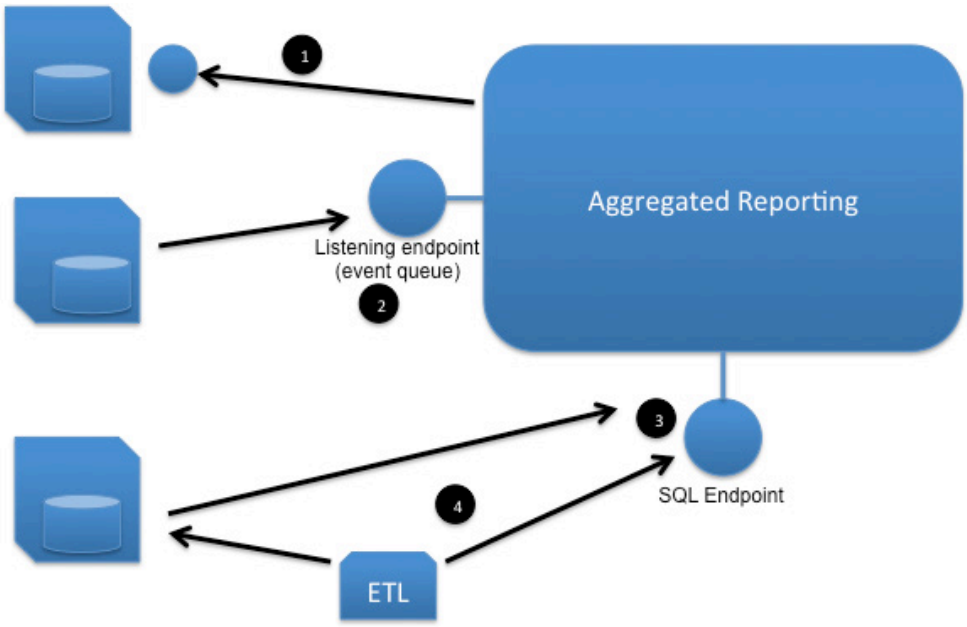


Figure 7.13 : Data sources for an aggregated reporting implementation. The illustration shows 4 ways to get data into an aggregated reporting implementation: actively going to the data (1), listening to events (2), SQL push by other services (3) or SQL push by ETL tools (4)

Essentially there are four ways

1. Actively calling other services – here the aggregated reporting will use other services' contract to occasionally sample them for new data. This is probably the worst way to go about it as the aggregated reporting implementation has to know about all the other services to be able to do that and the contracts should be expressive enough to export all the needed data
2. Passively getting data from services. There are actually two sub types here. One where services call the aggregated reporting server with data they wish to expose to reports e.g. by submitting a CSV file with exported data to the aggregated reporting service API. The second variant is to have the aggregated reporting implementation subscribe to events published by other services
3. Service SQL Push- Where services export a view of internal data, establish a connection to the aggregated reporting landing database, create their own tables and save data for reporting there.
4. ETL SQL push – Similar to 3, where the responsibility of getting data from services and getting it to the aggregated reporting is on an external tool. This isn't recommended, as mentioned above, as the ETL is likely to violate the services' autonomy to get the data. Form the aggregated reporting side, it is still ok as the ETL does not know the internal implementation/ representation of data within the service.

Ok, so we have the data in. What happens now? Figure 7.14 below illustrates the process that data goes through once it is arrives at the aggregated reporting service. In essence what happens now is Transform and Load process. Step one is the service side of what we've explained above. The SQL endpoint is recommended to use a landing database which is different from the raw storage store to provide security buffer.

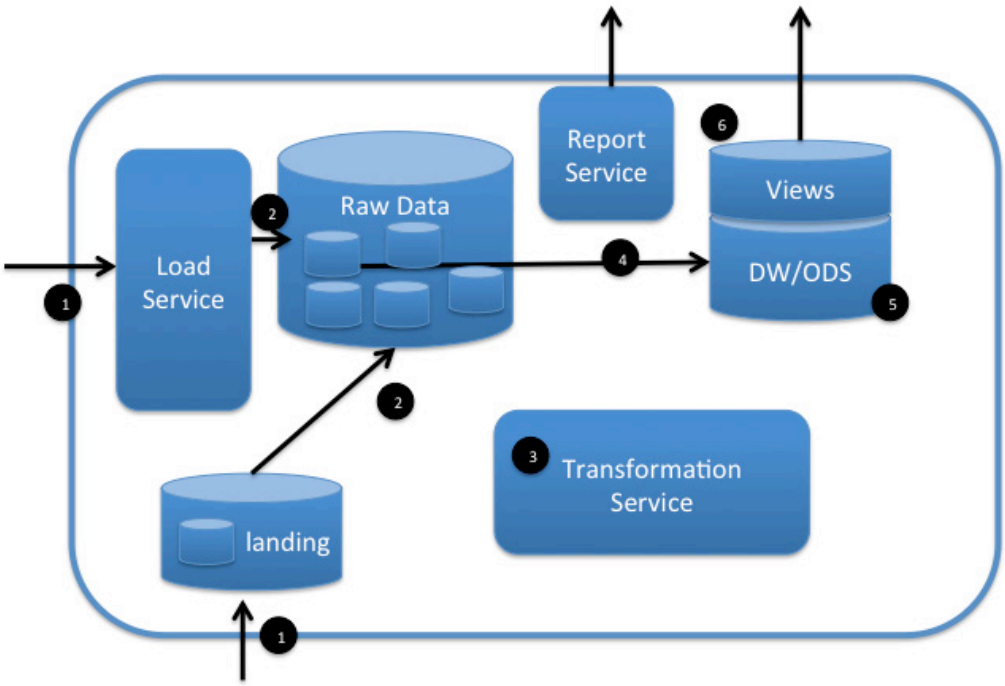


Figure 7.14 : data processing within the aggregated reporting pattern. First (1) we accept the data from external sources. Save it to a raw data store (2). Process it (3) and prepare it for reporting (4). The data is then saved into a Datamart or and ODS (6) and exposed (6) via a reporting interface and/or views

Step two is to get the data into the raw data store. There are two options here. One is a transient store, where the idea is to synchronize different data sources that arrive unsynchronized (each service can send its data independently). The other is to keep the raw data and build a big data store of data over time.

The next step is to process data, cleanse , aggregate, and prepare the data for reporting e.g. create star schemas, build cubes etc. Again, like in the previous stage we can make a choice between longer term and shorter term solution. If we opt for a shorter term solution we can set the reporting database as an operational data store (ODS) i.e. a database that is structured like a datamart (denormalized data)but with short retention. The second option is to create a datamart for reporting. It is more common to store the raw data long-term when choosing to use a datamart for reporting.

The last step is to expose the reports. This is done both via service interface for queries and predefined reports and SQL end points that exposes views that server as a contract and edge component to isolate the internal datamart structure from consumers.

Now that we understand better what aggregated reporting is, there are still a few open questions we need to address.

- How is aggregated reporting SOA friendly?
- How is better than SQL directly to each service
- How is it different than "entity aggregation"?

HOW IS AGGREGATED REPORTING SOA FRIENDLY?

how can the aggregated reporting get data from multiple, if not all, the services and not violate SOA principles. Well, what makes Aggregated Reporting a service is that the data it holds is immutable and the aggregated reporting is not the owner of changes in the data. It holds a representation of un-changing data for use in the service it provides (i.e. reporting). It is recommended, in this regard, that data kept by the aggregated reporting service whould be idempotent (e.g. versioned) so that the relations it expresses will always be true (for the versions involved). In any event, the source of "truth" are the original services whose data is mirrored. On the structural level the Aggregated reporting service is SOA compatible as it externalizes it capabilities via well-defined interfaces. The incoming SQL endpoint needs to be configurable via the regular service API i.e. a service should contact the service API to request an allocation of space, it will then be garanted a connection credentials to its own "landing zone". The implementation specifics can be different but the idea that the interaction with the incoming SQL endpoint is to be controlled via a contract should hold. As for the output SQL endpoint, here the separation of internal data structures and notion of contract should be implemented by a layer of views. The views represent the external agreement and the internal implementation don't have to match and can vary from the external one.

HOW DOES AGGREGATED REPORTING DEFER FROM DIRECT ACCESS TO EACH SERVICE INTERNAL DATABASE?

Firstly, as already mentioned the internal structure of a service might not be an RDBMS. Secondly, exposing SQL at each and every service increase the risk this will not be done right – either by exposing internal data structures, mangling up security etc. . The real benefit, however is that the aggregated reporting internal structure is built for reporting and as such it will, most likely provide much better performance than accessing even a single service directly as service's internal data stores are transaction (OLTP) oriented and not reporting oriented. – That is even more true for reports that need to be cross services.

HOW IS AGGREGATED REPORTING DIFFERENT FROM ENTITY AGGREGATION?

I included this question because it sounds they are similar as both have the word aggregation in their name. However the similarity ends there. Aggregated reporting keeps data ownership at the different services, is not focused on a single entity, is built on immutable data and is geared only towards reporting.

"WHAT ARE THE DRAWBACKS OF USING AGGREGATED REPORTING?"

I personally believe aggregated reporting is the best way to handle reporting in service oriented architecture. However like every design decision it does come with its own tradeoffs. The main tradeoffs here are the relative complexity of the solution (vs. reporting

going to each service) which translate to higher time to market; increased latency in terms of freshness of data (data has to be processed before it is available) and increased storage costs form duplication of data. The benefits, as we've already mentioned are high performance of reports, cohesive view of the data, separation of responsibilities, and keeping SOA's flexibility benefits. Additional non SOA benefits of the Aggregated Reporting pattern is the promotion of concepts such as Command-Query Responsibility Separation (CQRS) and Master Data Management (see further reading for more info on both).

7.3.3 Technology Mapping

We've explored the structure of the aggregated reporting pattern in the previous section and saw that it is has a lot of functionality, which means that there are plenty of ways to implement it and plenty of technologies that can play the various parts. As usually, the point of the technology mapping section is not to provide an exhaustive list of implementation option but rather provide a taste of what's possible. One of the interesting options, which grows in popularity in recent years is to implement aggregated reporting as a big data store (see also discussion on big data & Hadoop in chapter 10).

Recent systems I worked on used Hadoop and conventional datamart together as the aggregated reporting implementation. The Hadoop system was used as a datawarehouse with the long term, never delete, storage of historic data coming from all the services. The data from each of the services was saved in almost raw form in Hadoop's distributed file system (HDFS) as it arrived and processed at later interval to provide data useful for reporting. An ETL process took recent (last few months) of that data and exported it to a star schema in a conventional datamart (Oracle).

An interesting scenario in this regard is the case when the data exported to the datamart is summary data and not the detailed data. For instance, returning to the example at the beginning of this chapter, if we have the reps monthly performance data in the datamart and their call-by-call performance data stays in the datawarehouse.

Figure 7.15 below illustrates how a drill-through from datamart data to Hadoop data can occur. The first step occurs when the summary data is calculated (step 1). The map reduce job that calculates the summary and exports the data to the datamart also saves the origin of each calculation in HBase (HBase is an Hadoop based NoSql solution which support high-throughput random read/write)

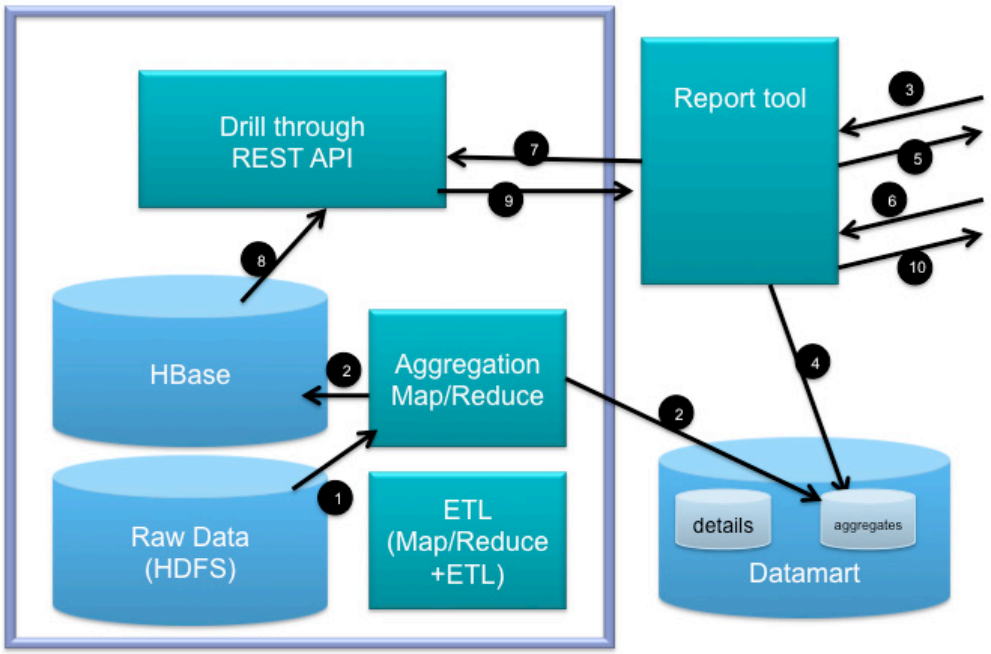


Figure 7.15 : Example for a drill through from summary data in a data mart to an Hadoop based data warehouse. When calculating summary data (1). The job writes the summary into the data mart and the source data to hbase (2). When a report is processed it runs against the datamart, (3,4,5). When a user requests to see how the summary data was calculated (6) the reporting tool makes a REST call (7) to a service which get the details from hbase (8) and provides it (9) to the reporting tool. Which in turn nories the user (10)

When a user runs a report on the aggregated data (step 3 4,5 in the diagram) she can see the data as it presented from the datamart (our sql endpoint in the aggregated reporting pattern) when the user wants to see the drill-through into each call that produced the score (step 5). The reporting tool then makes a call to the service endpoint of the aggregated reporting providing the key of the summary data (step 7). The service then accesses the list of sources generated when the summary data was generated (steps 8,9) and the data is presented to the end user (step 10).

Again, this is just one possible implementation. For instance, in another, smaller project we just used an operational data store to hold the latest data in a start schema with out retaining long-term historic view of the data. The details change but the architectural principles stay the same.

The last thing to discuss about the aggregated reporting pattern is "quality attributes" where things are a little different from other patterns in this book

7.3.4 Quality Attributes

The aggregated reporting pattern is probably the only architectural pattern in this book whose main drivers are functional requirements rather than architectural qualities. The reason the aggregated reporting pattern is still architectural is that its implications are solution/system wide and not local. As mentioned above, the aggregated reporting pattern provides a functional solution that still retains SOAs architectural benefits and that's its strength.

> **NOTE:** that the aggregated reporting pattern does promote desirable quality attributes like flexibility and maintainability but it isn't driven by them – its motivation, as mentioned above is functional and not non-functional.

7.4 Summary

Chapter 7's goal is to highlight the main integration patterns that enable services to work together and become a system rather than a bunch of services or a knot of un maintainable mess (see knot anti-pattern in the next chapter).

The chapter covered the following patterns:

- Servicebus pattern- allows connecting services in a loosely coupled manner.
- Orchestration which deals with externalizing business processes flow from services to promote flexibility and governance.
- Aggregated Reporting that provides a SOA-friendly way to solve the reporting conundrum

Chapter 7 ends part one of the book. The next part takes a look at some of the aspects of implementing SOA in the real world. The next chapter, the first of part two, will take a look at some of the common pitfalls, or anti-patterns that can ruin a fledging SOA implementation on the get go.

7.5 Further reading

Enterprise Integration Patterns, Gregor Hohpe & Bobby Woolf
http://www.eaipatterns.com/MessageBus.html
http://martinfowler.com/bliki/CQRS.html
http://www.cqrsinfo.com/
http://en.wikipedia.org/wiki/Master_data_management
http://www.amazon.com/MASTER-DATA-MANAGEMENT-GOVERNANCE/dp/0071744584/ref=pd_vtp_b_2
http://arnon.me/2010/09/soa-contracts-events-ownership/

8

Service Anti-patterns

The first part of the book looked at common challenges we meet when designing Service Oriented systems and provide contextual solutions (you know…the patterns), to help solve these challenges. The patterns are somewhat "synthetic" in the sense that they work well under context they were defined in. We know that the "real-world" is not synthetic and that boundaries and contexts will probably not be as clear cut as we'd like. The aim of the second part of the book is to cover some of the aspects of working in this less-than-perfect real world. The second part includes a case study that describes a system where quite a few of the patterns described in this book where implemented. The case study demonstrates how several patterns can play together to provide a complete solution. The chapter after that, takes a look at some "dos and don'ts" and covers some guidelines for handling things like versioning or simple pitfalls in implementing services. The first chapter of the second part, takes a look at SOA anti-patterns

Anti patterns are basically, the other side of the equation namely, instead of contexts and solutions we have common pitfalls we are likely to stumble upon and how to avoid and/or refactor them. Taking this complementary view (of pitfalls to solutions) is important as starting out with SOA it is easy to make these mistakes even when you follow guidance such as provided by the patterns mentioned earlier.

Discussing challenges in using anti-patterns is the natural choice for this book, as anti-patterns, like patterns are about contextual wisdom. In other words, anti-patterns talk both about when a behavior is a problem and when that behavior might be acceptable. The structure of the anti-patterns bears some resemblance to the discussion of patterns, however there are a few necessary changes. The discussion of an anti-pattern is as follows:

- Context and problem – present background and introduce the anti-pattern
- Consequences – What's wrong – why is the anti-pattern indeed a problem

- Causes – Why does the anti-pattern occur?
- Solutions – How to change the design to avoid the pitfalls/problems the anti-pattern causes.
- Known Exceptions – When following the anti-pattern might be acceptable

Table 8.1 below summarized the anti-patterns discussed in this chapter

Table 8.1 The list of service anti-patterns we discuss in this chapter

Anti-Pattern name	Problem highlighted
The Knot	A knot is an Anti-pattern where the services are tightly coupled by hardcoded point-to-point integration and context specific interfaces
Nano-services	Nonoservice is an Anti-pattern where a service is too fine grained. Nanoservice is a service whose overhead (communications, maintenance etc.) out-weights its utility.
Transactional Integration	Transactional Integration is an Anti-pattern where transactions extend across services boundaries (i.e. not isolated inside services)
Same Old Way	Same old way is an anti-pattern where we do the same things we did previously j dressing them in SOA clothing but Dressing what-ever we did before in SOA clothing

The first anti-pattern we'll take a look at is the Knot. It is an anti-pattern of SOA naiveté. Nevertheless we must understand its root causes so that we don't repeat it even as SOA veterans.

8.1 The Knot

Everything starts oh so well. Embarking on a new SOA initiative the whole team feels as if it is pure green field development. We venture on - The first service is designed. Hey look it got all these bells and whistles; we are even using XML so it must be good. Then we design the second service, it turns out the first service has to talk to the second – and vice versa. Then comes a third, it has to talk to the other two. The fourth service only talks to a couple of the previous ones. The twelfth talks to nine of the others and the fourteenth has to contact them all – yep our services are tangling up together into an inflexible, rigid knot

The above scenario might sound to you like a wacky and improbable scenario - why would anyone in their right mind do something like that? Let's take another look, with a concrete example this time and see how the road to hell is paved with good intentions. In Figure 8.1 below we see a vanilla ordering scenario. An ordering service sends the order details to a inventory service, where the items are identified in the inventory, marked for delivery and then sent to a delivery service which talks to external shipping companies such as DHL, FedEx etc.

Figure 8.1 A vanilla ordering scenario. An ordering service sends the order to a inventory service, which provisions the goods to a delivery service which is responsible to send the products to the customer

If we think about it more we'll see that when an item is missing from the inventory we probably have to talk to external suppliers, order the missing items and wait for their arrival- so the whole process is not immediate. Furthermore since the process takes time, it seems viable to cancel the process if an order is cancelled. It seems we have two options (see Figure 8.2) either the ordering service will ask the two other services to cancel processing related to the order or the two services call the ordering service before they decide what to do next. Naturally the system wouldn't stop here, we would want to introduce more services and more connections e.g. an Accounts Payable service that interacts with the external suppliers, the inventory service and the delivery service(since we also need to pay shipping companies) etc.

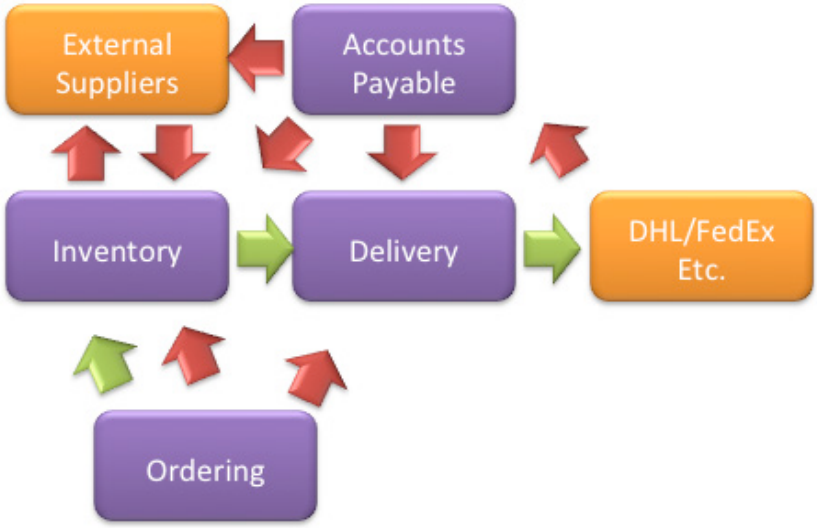


Figure 8.2 A little more realistic version of the Ordering scenario from figure 8.1. Now we also need to handle missing items in the inventory, cancelled orders and paying external suppliers. In this scenario the services get to be more coupled. For instance the Ordering service is now aware of the delivery service and not just the inventory service.

With each new service we draw more lines going from service to service, and with each new service we update the services' business logic with the new business rules as well as knowledge of the other services' contracts.

8.1.1 Consequences

Well, so we get more lines going from service to service that normal isn't it? After all if the services won't talk to each other they won't be very useful? Isn't that the whole point of SOA?

Well, yes – and no. Yes it is normal for services to connect to each other. After all, creating a system in an SOA is connecting services together. As for the "no" part, the problem lies with the way we develop these integrations if you are not careful it is easy to get all the integration lines in a big, ugly mess – a knot

A knot is an Anti-pattern where the services are tightly coupled by hardcoded point-to-point integration and context specific interfaces

For instance, what happens when we want to reuse the ordering service mentioned above. No problem, we just call it from the new context. Alas, the knot prevents us from reusing it without hauling in the rest of the baggage - all the other services we defined above (the inventory, delivery etc.) if the new context is not identical in it ordering processes and matches what we already have we can't use it. Or we can't use it without adding one-off interfaces where we add specific messages for the new context and all sort of "if" statements to distinguish between the old and the new behavior. Another option is to make this distinction in the original messages, which either not possible or forces us to make sure the other services are still functioning. In any event it is a big mess.

Let's recap. We moved to SOA to get flexibility, increase reuse/use within our systems, prevent spaghetti point to point integration – what we see here is not flexible , hard to maintain and basically it seems like we are back at square one and we invested gazillions of dollars to get there.

8.1.2 Causes

How did that happen? How can a wonderful, open standards, distributed, flexible SOA deteriorate to an unmanageable knot?

It is tempting to dismiss the knot as the result of lack of adequate planning. If we only planned everything in advance we wouldn't be in this mess now. Well, besides the point that trying to plan everything ahead of time is an anti-pattern in itself (an organizational anti-pattern – which isn't in the scope of this book). There's still a good chance you'd get to a Knot anyway since the problems are inherent in the way business work.

If we take a look back at the Integration Spaghetti scenario discussed in chapter 1 (depicted as figure 8.3 below), we can see that the phenomena was there as well, when our business processes evolve we find we need to interact with information from other parts of

the system. The flow of a business process expands to supply that needed information or service and thus the Knot grows.

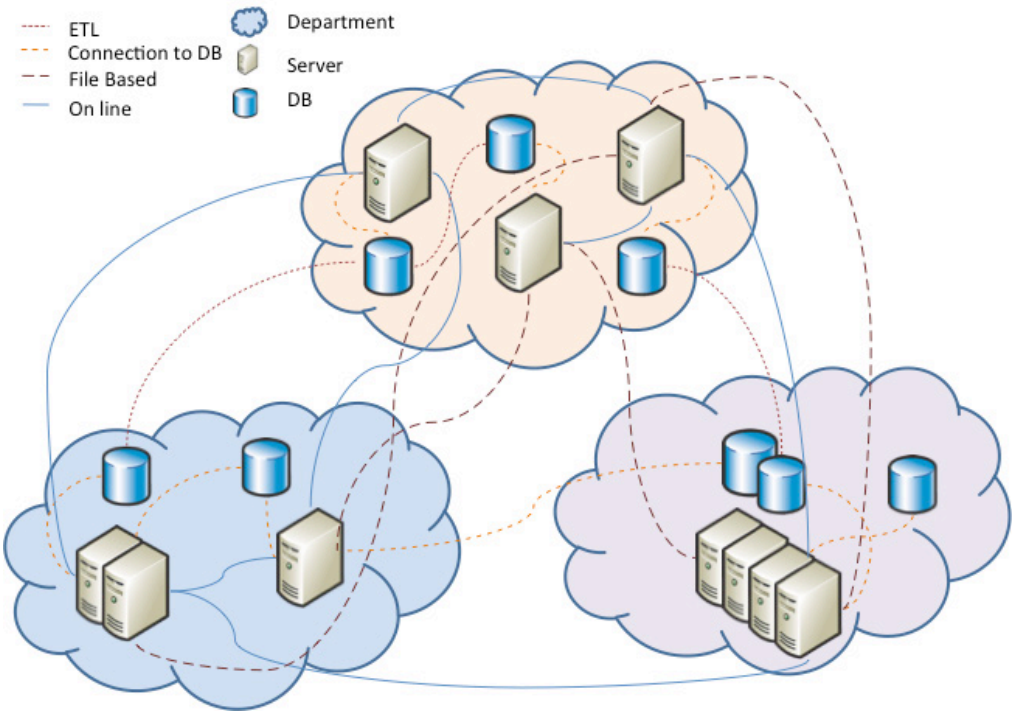


Figure 8.3 The Knot anti-pattern is similar in both effect and origin to the spaghetti integration in non-SOA environments

From the technical perspective, we have two forces working here. One is the granularity of the services. On the one hand, Services are sized so that a business process requires several of them to work together. On the other hand they aren't small enough so that they would be an end-node in the process (i.e. only other services would call the service and it will just return a result). Note that this isn't a bad thing in itself, after all if each process was implemented by a single service we'd have silos not unlike the ones we try to escape by using SOA and if we set the services too small we'd fall into another trap (see the Nanoservices anti-pattern later in this chapter). The bottom line is that while the granularity is a force that drives us toward the Knot, there's not a lot we can do about it without getting ourselves into worse problems.

The second, stronger, force that pushes a system into a Knot is the business process itself. Since, as we mentioned above, the process flows through the services, the services needs to be aware of the flow and then call other services to complete the flow. In order for

a service to call another service it has to know about its contract and know about its endpoint. When another business flow goes through that service we not only add the new contracts and endpoints but also the contextual knowledge of which other services to call depending on the process. And that's my friends, is exactly the thing that gets us into trouble – the services start to tie themselves to each other more and more, as we implement more business process and more flows.

Hey, you say, but SOA should have solved all that, surely there is something we can do about it – or is there?

8.1.3 Refactoring

The previous section explains that most of the problem is caused by having the services' code determine where to go next and what to do with the results of the services' processing. If there was only a way to somehow pry these decisions away from the services' greedy hands… As you'd probably guessed there is such away, in fact there are several such ways and this book lists three of them: The Workflodize pattern (Chapter 2), Orchestration (Chapter 7) and Inversion of Communications (Chapter 5). Let's take a brief look at each of these patterns and see how they help.

The workflodize pattern suggests adding a workflow engine inside the service to handle both Sagas (i.e. long running operations, see chapter 5) and added flexibility. The "added flexibility" is the card we want to play here. When we express the connections as steps in the workflow they are not part of our services' business logic. They are also easier to change in a configuration-like manner both of these points are big plusses.

Still, a better way to solve the service to service integration problem is to use an external orchestration engine. The idea of using the Orchestrated Choreography pattern is to enable Business Process Management- or a way for the organization to control and verify it processes are carried out as intended (you need an orchestration engine for that but it helps…). In the context of solving or avoiding the Knot anti-pattern, Orchestrated Choreography is better than Workflodize since it centralizes and externalize all the interactions between services and thus effectively removing all the problematic code from the services themselves. Note that there's a fine line between externalizing flow and externalizing the logic itself (see discussion in Orchestration pattern, in chapter 7).

The third pattern we can use to refactor the Knot is Inversion of Communications. Inversion of Communications means modeling the interactions between services as events rather than calls. Inversion of communications is, in my opinion, the strongest countermeasure to the knot. The two patterns mentioned above bring a lot of flexibility in routing the messages between the services. The inversion of communications pattern also helps the message designers remove specific contexts from the messages since when the service's status is raised as an event it isn't addressed to any other service in particular. Note that using inversion of communications doesn't negate using either of the two other patterns mentioned above since that once the event is raised we still need to route it to other services and using a workflow engine is a good option for that. Another implementation

option is to use an infrastructure that support publish/subscribe (see the pattern's description in chapter 5 for more details.)

Going back to the ordering scenario we mentioned above. As I mentioned, the services grow with needless knowledge of specific business process. So for instance, the ordering service had to know both about the inventory service and the delivery one. Refactored with the Inversion of Communications pattern, the same Ordering service, doesn't have to know about any of the other services. In Figure 8.4 we can now see that the Ordering service sends two business events (new order, cancelled order) and the routing of these messages is no longer the responsibility of the service

Figure 8.4 The Ordering service using the Inversion of Communications pattern. Now the service doesn't know/depend on other services directly. It is only aware of the business events of new order and cancelled order which are relevant to the business function that the service handles

Refactorings aside, one question we still need to think about is whether there are any circumstances where having a Knot is acceptable.

8.1.4 Known Exceptions

In a sense the Knot is a distributed version of an anti-pattern described by Brian Foote and Joseph Yoder as "Big Ball of Mud" – spaghetti code where different types of the system tied to each other in unmanageable ways. The reason for mentioning the connection is that the reason that "Big Ball of Mud" might be considered a pattern rather than an anti-pattern also apply here:

"[when] You need to deliver quality software on time on budget… focus first of feature and functionality, then focus on architecture and performance"

Starting out on a large project, such as moving an enterprise to SOA, is difficult enough as it is. You can't figure everything in advance, you need to deliver something – so as the Nike slogan goes "just do it". Get something done. You do need to be prepared to let go and redesign further down the road. In the current system I'm working on – a visual recognition/search engine for mobile, we went with a "knot" approach for the first release. The simplicity of the implementation, i.e. less investment in infrastructure, ad hoc integration etc. enabled us to deliver a first working version in less than 6 months. These 6 months also helped us understand the domain we are operating in much better and more importantly get to market with the feature the business needed in the schedule the business wanted. We spent the next 6 month rewriting the system in a proper way, including applying the Inversion of Communications pattern mentioned above.

To sum this up, coding the integration code into services is likely to end as a Knot. It is acceptable to go down this path for a prototype or first version i.e. to show quick results. However you do need to plan/make the time to refactor the solution so you will not get stuck down the road

One of the forces contributing to the forming of the Knot was the granularity of services. The next anti-pattern talks about another granularity related problem. Nanoservices – sometimes size do matter

8.2 Nanoservices

There are many unsolved mysteries, you've probably heard about some of them like the Loch Ness monster, Bigfoot etc. However, the greatest mystery, or so I've heard, is getting the granularity of services right… Kidding aside, getting right-sized services is indeed one of the toughest tasks designing services – there's a lot to balance here e.g. the communications overhead, the flexibility of the system, reuse potential etc. I don't have the service granularity codex and deciding the best granularity depends on the specific context and decisions (e.g. the examples in the Knot anti-pattern above). It is an easier task to define what shouldn't be a service for instance, calling all of your existing ERP system a single service should definitely be shunned. The Nanoservices anti-pattern talks about the other extreme… the smaller services

Consider, for instance, the "calculator service" which appears in samples web-wide (I've personally seen examples in .NET, Java, PHP, C++ and a few more). A basic desk calculator, as we all know, supports several simple operations like add, subtract, multiply and divide and sometimes a few more. Implementing a calculator service isn't very complicated. Consider for example, Listing 8.1 below taken from an apache sample, that shows part of a WSDL of a java calculator service that, lo and behold, accepts two numbers and adds them.

Listing 8.1 excerpt from a WSDL of a stateless "calculator service

```
<wsdl:types>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns="http://jws.samples.geronimo.apache.org"
```

```
targetNamespace="http://jws.samples.geronimo.apache.org"
                     attributeFormDefault="unqualified"
elementFormDefault="qualified">

             <xsd:element name="add">
                <xsd:complexType>
                   <xsd:sequence>
                      <xsd:element name="value1" type="xsd:int"/>
                      <xsd:element name="value2" type="xsd:int"/>
                   </xsd:sequence>
                </xsd:complexType>
             </xsd:element>

             <xsd:element name="addResponse">
                <xsd:complexType>
                   <xsd:sequence>
                      <xsd:element name="return" type="xsd:int"/>
                   </xsd:sequence>
                </xsd:complexType>
             </xsd:element>
          </xsd:schema>
    </wsdl:types>

    <wsdl:message name="add">
       <wsdl:part name="add" element="tns:add"/>
    </wsdl:message>

    <wsdl:message name="addResponse">
       <wsdl:part name="addResponse" element="tns:addResponse"/>
    </wsdl:message>

    <wsdl:portType name="CalculatorPortType">
       <wsdl:operation name="add">
         <wsdl:input name="add" message="tns:add"/>
         <wsdl:output name="addResponse" message="tns:addResponse"/>
       </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="CalculatorSoapBinding"
type="tns:CalculatorPortType">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

        <wsdl:operation name="add">
            <soap:operation soapAction="add" style="document"/>
            <wsdl:input name="add">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="addResponse">
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>

    </wsdl:binding>

    <wsdl:service name="Calculator">
```

```
        <wsdl:port name="CalculatorPort"
binding="tns:CalculatorSoapBinding">
            <soap:address location="http://localhost:8080/jaxws-
calculator/calculator"/>
        </wsdl:port>
    </wsdl:service>
```

Calculator services can be even more advanced and have memory - Listing 8.2 below taken from an MSDN sample shows the interface definition for such a calculator in .NET. it is a WCF sample that uses workflow services and accepts a single value at a time

Listing 8.2 a Service contract definition for a statufil calculator service

```
[ServiceContract(Namespace = "http://Microsoft.WorkflowServices.Samples")]
public interface ICalculator
{
    [OperationContract()]
    int PowerOn();
    [OperationContract()]
    int Add(int value);
    [OperationContract()]
    int Subtract(int value);
    [OperationContract()]
    int Multiply(int value);
    [OperationContract()]
    int Divide(int value);
    [OperationContract()]
    void PowerOff();
}
```

The calculator service (both versions of it) is a very fine grained service. Naturally, or hopefully anyway, the calculator examples are just over simplified services used to demonstrate SOA related technologies (JAX-WS in the first excerpt and WCF and WF in the second one). The problem is when we see this level of granularity in real life services

8.2.1 Consequences

Problem? Why is "fine granularity" a problem anyway? Isn't SOA all about breaking down monolith "silos" into small reusable services? More so, the finer grained a service is, the less context it carries. The less context a service carries the more reuse potential it has – and reuse is one of the holy grails of SOA isn't it? The calculator service above seems like the epitome of a reusable service. There's no doubt we can reuse it over and over and over.

Reuse is indeed a noble goal (I'll leave discussing how real it is for another occasion), the culprit of fine grained services, however, is the network. Services are consumed over networks – both local (LANs) and remote (extranets, WANs etc.). The result is that services are bound by the limitations and costs incurred by those networks. Trying to disregard these costs is exactly what ailed most, if not all, RPC distributed system approaches that predated SOA (Corba, DCOM etc.) - The calculator service and other similarly sized services are nanoserivces.

Nonoservice is an Anti-pattern where a service is too fine grained. Nanoservice is a service whose overhead (communications, maintenance etc.) out-weights its utility.

So how can nanoservices harm your SOA? Nanoservices cause many problems, the major ones being poor performance, fragmented logic and overhead. Let's look at them one by one

Every time we send a request to a service we incur a few costs such as serialization on caller, moving caller process to the OS network service, translation to the underlying network protocol, traveling on the network, moving from the OS network service to the called process, deserialization on the called process – and that's before adding security (encryption, firewalls etc), routing , retries etc. Modern networks and servers can make all this happen rather fast but if we have a lot of nanoservices running around these numbers add-up to a significant performance nightmare,

Nano-services cause fragmented logic - almost by definition. As we break what should have been a meaningful cohesive service, into miniscule steps our logic is scattered between the bits that are needed to complete the business service. The fact that you need to haul over several services to accomplish something meaningful also spell increased chances of the Knot anti-pattern, mentioned above.

Proliferation of Nanoservices also causes development and management overhead. Just look at the amount of WSDL needed to define the calculator services in listing 8.1 above and for what? A service that adds a couple of numbers… There is a relatively fixed overhead associated with managing a service. This include things like keeping track of a service in a service registry, making sure it adheres to policy, writing the cruft (things we have to write around the business logic) for configuring it etc. Having nano-services around means we have to do this a whole-lot more times (i.e. per service) compared with having fewer coarser grained services.

The point of overhead out-weighing utility that appears in the Nano-services definition above is subtle but important. The fact that a contract does not have a lot of operations means we want to make sure we don't have a nanoservice, but it doesn't automatically mean that it is. For instance, a fraud detection service contract might only accept transaction details and decide whether to authorize the transaction, deny it or move to further investigation. However the innards of this service involve a complex process like running the details in a rule engine checking for fraudulent behavior patterns, matching to black lists etc. In fact Fraud detection is such a complicated issue that these are actually systems and a SOA based one would be comprised of several services in itself.

The other side of the equation is also true a comprehensive contract does not guarantee a service is not a nano-service. For instance, in a system I designed on the initial iterations we developed a resource management service. It supported some very nice operations like getting status of all the services in the system, running sagas and of course allocating services. Allocating services meant that whenever an event went out that needed a (new)

service instance to handle it, we had to make a call to the resource manager to get one. This provides for a neat centralized management and also for a performance bottleneck that slows the whole system. To solve this we went with distributed resource management but that's beyond the scope of this discussion. The point, however that is that the utility of the resource management (e.g. easy management of running sagas) vs. overhead associated with the service (the number of calls and performance hit on the system) was not worth it – Hench a nano-service.

8.2.2 Causes

From a more technical point of view, we get to nanoservices from not paying attention to at least a couple of the fallacies of distributed computing. Mentioned in chapter 1, the fallacies of distributed computing are a few false assumptions that are easy to make and prove to be wrong and costly down the road. Specifically, we are talking here about assuming that

Bandwidth is infinite – Even though bandwidth gets better and better, it is still not infinite within a specific setup. For instance in one project we were sending images over the wire and distribute them to computational services (a la map/reduce – see also Gridable Service in chapter 3). Things were working ok when we sent small images, but when we sent larger images we understood we were sending them as bitmaps and not as much more compact jpegs which caused a burden on the backbone of our switches which wasn't ready for that load.

Transport cost is zero – As explained in the previous section every over-the-wire call incurs a lot of costs vs. a local call (also see figure 8.5 below).The costs of the transport can be considered both from the time it take to make each of these calls but even the real dollar value attached to making sure you have enough bandwidth (connection/routers, firewalls) to handle the traffic incurred

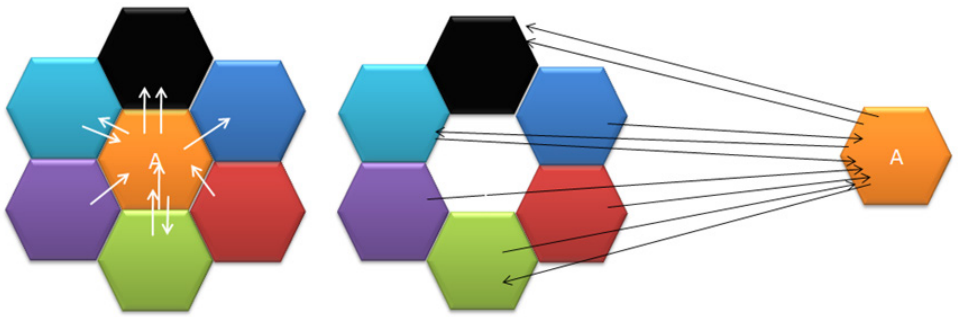


Figure 8.5 Local objects can "afford" to have intricate interactions with their surroundings. A similar functionality delivered over a network is more likely than not to cause poor performance because of the network related overhead.

Another reason to get Nano-Services, at least for beginners are poor examples – as, noted the calculator services above are taken from real examples provided by various vendors. SOA newcomers and/or people without a lot of distributed systems development experience can be easily take these samples at face value, and go about implementing services with similar granularity. The fact that that web-service framework mostly map service calls to object method calls makes this even more tempting.

Nano-services is also an inherent risk when applying the orchestrated choreography pattern. Adding an orchestration engine, capable of controlling flow and external to services tempts us to think that we can use it to drive all flow as little as it may seem. Couple this with the fact that the smaller the services are the more "Reuseable" they are (less context) and, again, you may end up with a lot of nano-services on your hands.

Lastly, since the nano-services boundary is soft (remember utility vs. overhead weight) behaviors that can look promising at design time can prove to be nano-services moving along (like the resource manager example above). This can be an acceptable if your SOA is developed iteratively (see 8.2.4 exceptions below) but it still mean that we have to come up with ways to refactor nano-services.

8.2.3 Refactoring

There are basically two main ways to solve the nano-services problem. One, which is relatively easy, is to group related nano-services into a larger service. The second option, which is more complicated, is to redistribute its functionality among other services. Let's take a look at them one by one.

On one project I was working on we needed to send out notifications to users and admins via SMS messages. Since the software component that did the actual SMS dissemination was a 3rd party app we've decided to create a simple service (not unlike OO adapter) that accepts requests for SMS and talks to the 3rd party software. A nano-service was born, it even got a nice little name Post Office Service (ok, ok the original name was Spam Server but I thought it would look bad in presentations ☺).

Why is this a nano service? Well, it really doesn't do much and it would be even simpler to package this as a library that other services can use and it does have all the management overhead of maintain as another system service.

What we did about it was to add similar functionality to the service so it also learned to send emails, tweets and MMSs. A serendipitous effect of this was that now instead of sending a request like TweetMessage or SendSMS to this service we could now raise more meaningful events such as SystemFailureEvent and have the service make decisions on how to alert administrators based on the severity of the problem etc. So combining the related functionality helped make the overall service even more meaningful.

Unfortunately it isn't always possible to take the functionality of Nano-services and find suitable "other services" (nano-or right-sized) that can assimilate them. In those cases getting rid of a nano-service is more of an exercise in redesign than is a refactoring. For instance, in a project we've built we had a services allocation service (SAS). The SAS role

was to know about other services location and health status and utilization and upon a request, such as beginning a Saga (see chapter 5 for the saga pattern) decide what service instances should be used. The service also provided "reporting" capabilities for active sagas, services utilization etc. This might not sound like a nano-service, and at first we thought so too, but as the project progressed we found that being a central hub, as seen in figure 8.6 below, made the SAS a performance bottleneck, incurring additional costs (in latency) on a lot of the calls and interactions made by other services. The utility of the SAS, of finding what service instance to talk to, was being diminished by the cost – yep it is a nano-service after all.

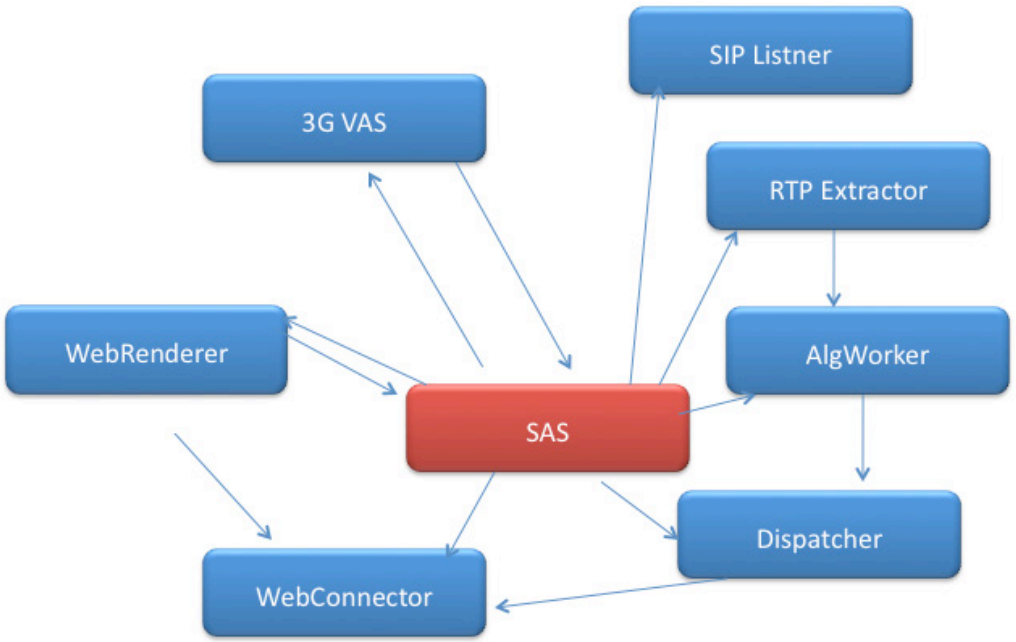


Figure 8.6. Sometime Nano services can provide important services but the costs of that functionality is more than their usefulness – here we have a synchronization service that became a bottleneck for performance.(everything goes through it)

To solve the SAS problem we had to put in quite a lot of work. The solution, essentially was to move to distributed resource management, so that each service had some knowledge of what the world looks like so that it could decide what service instant to talk to by itself.

To sum this section, sometimes it is easy to notice that something is a nano-services, chances are that in these cases it would also be easy to take the functionality and group it with related functionality in other services. However on other occasions the fact that a service provides too little benefit is not as apparent and only becomes clear as we move

along. In those cases it is also harder to fix the problem. One question we still need to cover is are there any situations where we would go with a nano-service even if we know it is one on the onset.

8.2.4 Known Exceptions

When is it ok to have Nano-services? When you are starting out. When your approach to SOA is evolutionary and you don't plan everything in advance (something that rarely work anyway, but that's another story), there's a good chance that first versions of services you build will not show a lot of business benefit, but they will already need the full overhead of a service. The post office service in the example above is a good example for that as starting out it only dealt with a single type of message and it didn't do a whole lot with it either.

The post office service is also a good example for another reason to have a nano-service which is when you want to build an adapter or bridge to other systems be that legacy systems or 3rd party ones. In these cases you need to weight the advantage of using a service vs. building the same functionality as a library that can be used within services, but in many cases keeping the flexibility and composability of SOA can triumph over the overhead associated with having an additional service to manage.

Lastly, one point to keep in mind is that NanoServices is a rather soft pattern and the value of a small service can radically change from system to system or even in a certain system as time and requirements progress. It is worthwhile questioning our assumptions and looking at the services that we grow from time to time to validate the usefulness of what we're building.

Another anti-pattern where people take their old know-how into SOA without considering the consequences is the transactional integration anti-pattern.

8.3 Transactional Integration

It all starts with a business requirement – as it always should. We have an ordering system (say the same one from the Knot anti-pattern) and the business says they only want to confirm an order to the user if the item is already secured for that order in the inventory. From the technical point of view we have 2 separate services - one handles orders the orders and the other handles inventory – now what?

Figure 8.7 A vanilla ordering scenario. An Ordering service needs to confirm item is available before confirming order for customer.

This sounds like a text book case for using transactions but in reality it isn't. I am going to explain why in a short while, but before we go there let's do a (very) short recap on transactions and distributed transactions.

Transactions basically build on four basic tenets: Atomicity, Consistency, Isolation and Durability (ACID):

Atomic – "All or nothing" meaning that once a transaction ends the state is either completely done (commit) or undone (abort)

Consistent – The actions included in the transaction are done together so the state is kept consistent. If you were to remove something from inventory and add it to shipment in the same transaction you won't have a situation where the item was removed from inventory and not added to shipping list

Isolated – while the transaction is in progress, logic which is not with part of the transaction will not see the world in its inconsistent form

Durable – The consequences of transaction are saved to persistent storage so that they are available after a system restart

The simplest way to get transactions is using "pessimistic locking". In this case a writer can only write if no other resource is reading or writing and a reader can only read if no other resource is writing (for a specific piece or block of data). On top of that to ensure ACIDness you need to write the data twice; once where you want it to end up and a second to a log file. This double bookkeeping ensures that if a crash occurs before the transaction was finalized (committed or aborted) you can check to see that both copies match and if not either (re)apply the log or rollback the data (this is somewhat a simplistic view –see further reading for more thorough explanations)

Unfortunately pessimistic locks rarely work in real life scenarios so more advanced ways of locking and still maintaining ACIDness were developed. However, all the mechanisms holds resources for the transactions and all locking mechanism build on the assumption that the time spent inside the transaction is short.

The plot thickens further, when it comes to distributed transactions. Now we have at least two transactional resources and not only do each of them has to handle the transaction we also need to coordinate the state between them – since if one commits the transaction and the other rolls it back the overall transaction is incomplete. Still computer scientist where smart enough to come up with several solutions to achieving distributed consensus and gave us two-phase commits, three-phase commits, paxos commits and whatnot. Case closed we can use transactions in SOA and life is beautiful.

Or is it?

8.3.1 Consequences

Well first off the transactions, even the distributed ones are not a problem in themselves. For instance chapter 2 introduced the transactional service pattern to allow handling

incoming messages in a reliable manner. The problems begin when the transaction scope involve more than one service or in other words:

Transactional Integration is an Anti-pattern where transactions extend across services boundaries (i.e. not isolated inside services)

So what sorts of problems can transactional integration introduce to your SOA? Quite a few actually, with the main three being Performance problems, Security threats and rigidity let's take a look at them one by one.

With all the goodness transactions does it also introduce temporal coupling i.e. the need for all the involved actions to finalize on or about the same time. Even if the locks held while the transaction enfolds are permissive (optimistic), the coordination that is needed to ensure consistency needs to be synchronized. When you develop a non-SOA you may be able to take all the performance considerations at design time and make sure the system behaves. I'd still say distributed transactions are not highly recommended even then, since the rigidity of the consistency needed when trying to achieve a distributed consensus can still mean holding locks for a long time in cases of partial failures.

The situation is much worse in an SOA since each service can and will evolve independently both in terms of deployment and functionality. For instance, what will happen when the inventory service moves to another datacenter (e.g. ported to the cloud). What if the designers of the inventory service decide that when the inventory level hits a thresh-hold they want to automatically order new supplies and they want that in a transaction – Now you cannot secure an item in the inventory until new supplies are ordered. All of a sudden our transaction expanded and now includes the ordering service, the inventory service and a supplier' service(s). So one risk is that designers of services participating in our transaction extend the transaction to handle business rules they need to comply with.

Another risk highlighted by the above mentioned scenario is related to security. In the example we added the suppliers' services into our transaction. This means that we now run the risk that external systems will now hold locks on our system, either maliciously or by neglect, effectively creating a denial of service scenario on our services. Service boundary, its edge, should also be a trust boundary, externalizing transactions to third parties might be far-fetched but externalizing it to other teams within the organization which work on their own services with their own priorities is not, the same risk applies there.

The last risk we can highlight with this example is connected directly to the Knot anti-pattern (which is one reason the same sample scenario is used) – Having transactions between services increases the coupling between them and increased coupling increases the risk of ending up with a Knot, which effectively kills SOA.

One can argue that most of even all of these are hypothetical situations and that when we design the SOA we can take the real constraints into consideration and plan for them – isn't that why we have enterprise architects for? Though the scenarios are over-simplified to illustrate the problems in a clear-cut way, real life scenarios manifest the same problem in

subtler ways. The main point is that evolvability and flexibility are the hallmark of SOA – That's why we want an SOA in the first place - so that we can evolve the IT of the organization to better match the **changing** needs of the business. The end result is that regardless of how we plan it out on the onset, in the long term it is hard to control who participate in the transactions which means that adding distributed transactions to the mix is an accident waiting to happen.

8.3.2 Causes

The main reason Transactional Integration happens was already mentioned above, when we start out and design our SOA we have a relatively good grasp on the enterprise business. Again, the problem is that an SOA solution is not static – if it is than it is probably a needless overhead to architect it as an SOA anyway.

Even if you do have a good initial understanding of the business flows, that understanding can deteriorate pretty fast. It isn't just that requirements change over time – an even greater force of change is getting deeper understanding as to what is exactly needed. To side track a little, the pragmatic way to implement an enterprise-wide SOA is not to do a multi-month (if not multi-year) project just to documents and design the overall architecture and services and only then begin the transition to the new architecture. Doing that is like sending a message that new things should just sit there and wait until you'd be ready – I am yet to see a business that can afford that. No, the more realistic and cost-effective way is to do some upfront design but also begin developing real services and work them into the existing software portfolio. This is sort of like building a new intersection where you also have to build detours, keep some of the lanes open - anything to keep the traffic going. When you work on an SOA in this manner the rework, the growing understanding of the business and the requirements changes means you can expect a lot of evolution to happen, as mentioned above, transactional integration will make that unlikely or very hard.

Other forces pushing to the transactional integration anti-pattern are the marketing organization of technology vendors. That's an odd statement so let me explain. Technology vendors provide, well, er technology. Thus when a vendor ships a technology frameworks, it is usually geared to answer a broad variety of needs. Take for example, Microsoft's Windows Communication Foundation (WCF) which is a unified infrastructure for remote communications between components. WCF offers message based communications along support for names-pipes; it is built to replace RPC technologies like remoting and it supports SOAP (WS*) web services, some support for REST style services and what not – yet WCF is by and large is marketed as an "SOA foundation". This isn't to say you can't use WCF for SOA but it does a lot more, it also does transactions… Other vendors follow the same path, whenever there's a new buzzword, their marketing organizations take whatever technology they currently have and slap that on it. The end result of that is a lot of confusion in regards to what right and what is not. The use of transactions for cross service integration is, unfortunately, just one sample of this effect.

Well, if transactions are not the way to go, what can we do instead?

Refactoring

There are several options to get around the problem either using Orchestration or Sagas, Inversion Communications pattern and similar means to achieve eventual consistency.

But wait, what exactly is the problem we're trying to solve? We are trying to achieve distributed consensus and consistency in the data and business picture as seen by several services. Let's take a quick look at the business scenario presented above. We have an ordering system and the business says they only want to confirm an order to the user if the item is already secured for that order in the inventory.

One way to solve this would be to externalize both the transaction scope and the business flow to an orchestration engine (see Orchestration pattern in chapter 6). The advantage over transactions directed from within the services is that the orchestration engine has the full picture both of the services involved (and their various trust levels) and the flow of which service calls to which service so there's more control on who does what and when. Still services participating need to be transaction aware and need to retain internal locks for external constraints so use this with caution.

Another alternative is to use sagas (see Saga pattern in chapter 5), Sagas basically means running a long running interaction (where messages are related and belong to the same "conversation) but without holding the same transactional guarantees as ACID transactions. For instance in case of a inventory problem the ordering service will have to do a compensating action to handle the error. In order for this to work in a reasonable manner the services may need to hold some data about the world such as a some data about inventory levels so it can receive a reasonable decision on its own.

Sagas can be augmented by the inversion communications pattern to make the services send events of their actions and other events to subscribe to them to create choreography scenarios. In our example the order service would publish that it has a new order that needs handling and the inventory service would listen on that. Once the inventory secures the items it will publish an event that says that so the Ordering service can notify the customer that the order is ready (maybe there would be additional steps like actually shipping the product etc.)

Both Sagas and the Inversion Communication pattern actually implement an eventually consistent system – we basically relax the temporal constraints on decision making by the various services. This can, and usually does, translate into how the business works in general. In our ordering example it may mean that it would be better to send an additional notification to the customer that her order was received, when the order service processed the order and sent the event to that affect

8.3.3 Known Exceptions

I can't think of a lot of SOA solutions that would benefit from cross service transactions. Transactional integration is usually a bad idea for most distributed systems anyway – from similar reasons to the ones mentioned above.

A rare exception to this rule might be for a closed solution (i.e. a system and not an organization) that is building on SOA principles. In a closed environment where everything is controlled it might be possible to pull it off without suffering from the rigidity and performance problems induced by Transactional Integration. Even in these rare cases it would still be preferable to control the transaction scope outside of the service by using an orchestration engine. Using orchestration means that at least the scope of transactions and the general flow of the business processes will be handled in the same place.

I would still be wary of going down this path since even closed control system tend to evolve over time so be forewarned.

A related anti-pattern, which bears some resemblance to Transactional Integration, is the Same-Old-Way anti-pattern. An anti-pattern of moving to a new architecture-style without fully understanding it.

8.4 Same Old Way

Every time a new concept makes headway, technology vendors' marketing departments run amok to brand whatever their current offering is, with the "shiny new thing". We've seen this phenomena occur over and over: with Agile, Cloud, Big Data and, yes, also with SOA. Savvy developers that we are, we're mostly smart enough to know that the first incarnation of a product on hype cycle is just that. However it is harder for us to notice when we do pretty much the same thing with our designs

Let's look at a simple example. Figure 8.8 below shows a sample "SOA" based architecture. On the left hand we have a "Data Service". A database wrapped in web-service or RESTful clothing. On top of it "Business Services" or "Entities" where the business logic of handling customers and accounts happens topped with CRUD service interface. What's missing is an additional tier which would be a UI consuming these so-called services – though most likely it would only see the Service Interface and not the other types

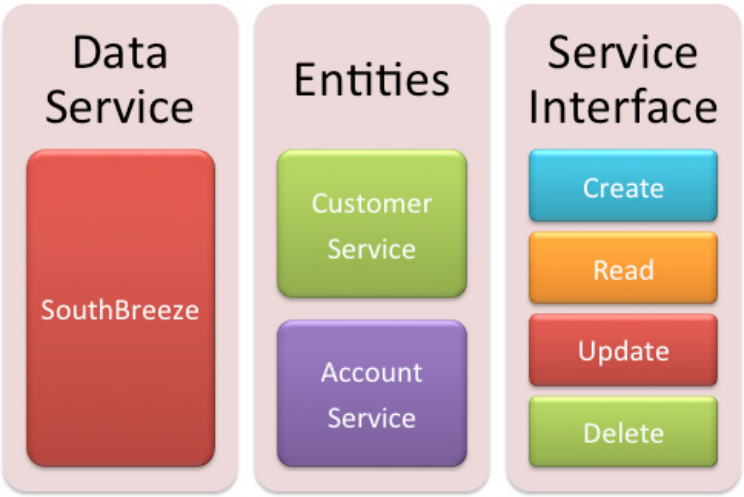


Figure 8.8 This is not an SOA – There are several components that have "service" names (the data service, customer service etc.) however the distribution of logic and component is does not follow SOA principles.

The next thing to do is take a look at what's wrong with this picture. However, I want to emphasize one point before we do that. You may think that I've just built a straw man here so it is no wonder it is going to be easy for me to destroy it (I guess that's somewhat true for the other anti-patterns as well) but these examples are based on things I actually saw in systems I reviewed. So, again, - let's look at what's wrong with this picture.

Well, the problem is Figure 8.8 does not describe an SOA – it is an n-layer/n-tier architecture. Sure the moniker "service" is thrown all over the place but if we examine more closely we can identify the layers. There's the data layer which is most likely a tier (layers are logical, tiers are physical) . The entities and the service interface will most likely reside on the application server tier and will be 2 layers within the business logic.

8.4.1 Consequences

So what? So we called a n-tier architecture an SOA. What's the big deal? If indeed, the only wrong thing in the sample was wrong names than there's no harm done (except maybe to SOA's name) The problem is that many times these layers and tiers are implemented as services – that is, they have separation between contract and implementation, they may run as autonomous services etc.

The problem, then, can be summed up as follows:

Same old way is an Anti-pattern where we implement non SOA architectures with SOA tooling and overhead – paying the SOA tax without reaping the SOA benefits

Let's clarify this a little. Not reaping SOA benefits means that, since it isn't in fact SOA you don't get the flexibility you wanted or the lowering the system level complexity by breaking the solution to smaller pieces.

SOA tax means that we have to invest more both in design time and in run-time. SOA means, for example, increased latency, since there are additional layers like serialization and deserialization, communications etc. If instead of two services we could manage using objects that talk to each other on the same memory space we'd have none of that. SOA tax can also mean the increase in local complexity of each component. For instance the implementation for the Data service above was something like Figure 8.9 below. We have a service hosted in a web server sporting a rich REST API that enabled queries and the other CRUD operations instead of just the data access layer we would have used otherwise. Not to mention the extra effort in testing, deploying and monitoring this service.

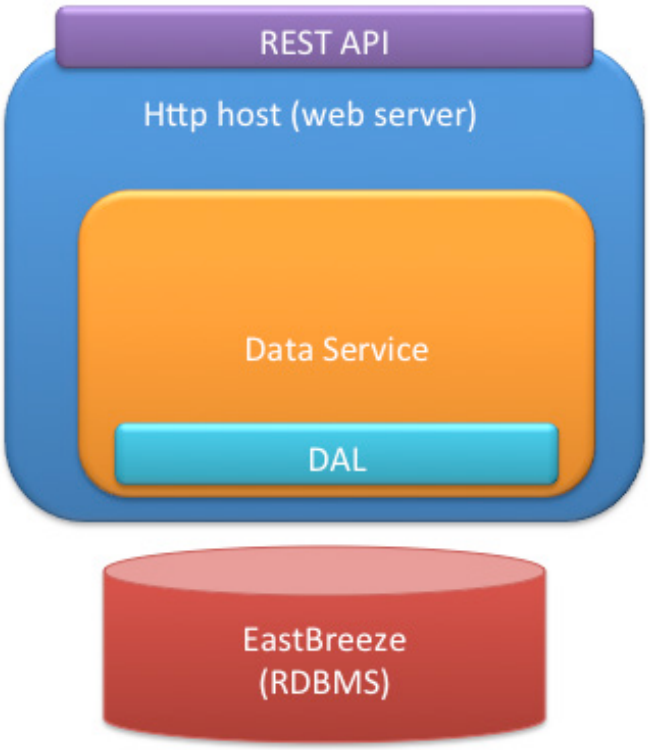


Figure 8.9 Structure of a Data service. A web based service host, hosting a service exposing a rich REST API for quering and updating an underlying RDBMS.

For all the work the data service needed, the benefit we get from using it (in the scenario above) over a simple DAL is nothing.

To look at all this from a broader perspective when we have a same-old-way anti-pattern on our hands the constraints of the real architecture hinder our ability to utilize SOA properly and the constraints of SOA hinder our ability to use the real underlying architecture effectively as well – and that's not a good place to find ourselves in.

8.4.2 Causes

Same-old-way is primarily caused by lack of SOA understanding. This ignorance is aided by confusion originated with vendor's pushing SOA related technologies as SOA itself. The most obvious of those is the, wrong, direct association between web-services and SOA services. Sure SOA services can be implanted using web-services. They can, however, be implemented with a messaging API, in conjunction with EDA (see Inversion of Communications in chapter 5), RESTful API, thrift etc. Not only that most of not all these other ways are better that web-services in many scenarios. Furthermore the technologies wrapping web-services (like WCF in the .NET world) are general purpose remoting mechanisms so that it is natural to use them outside of SOA anyway.

Another possible cause for a system to be a state matching same-old-way anti-pattern is if you have a system in transition from another architectural style to SOA. In which case, it is probably not an occurrence of the anti-pattern so much as an interim-state and the architects are not unaware of the state or the consequences.

8.4.3 Refactoring

The main "trick" with refactoring this anti-pattern is, probably, noticing that you are inflicted by it. I.e. acknowledging that we are indeed just bending whatever we're used to do into SOA clothing. To help you identify the problem you can think about the fallacies of distributed computing (see details in section 1.1.3 of chapter 1) – if you see that your SOA violates them than that is a smell you want to consider.

Unfortunately it isn't easy to refactor this anti-pattern. Not only will solving the problem will most likely result in a re-design and not a refactoring there's no straight recipe to get there. In essence you have to get a better understanding of SOA, it principles and constraints (hopefully this book helped with that) and they redesign accordingly.

If we look back at our over simplified problem above, Data Service can be ok if, for example, it is the API for your implementation of the aggregated reporting pattern (see chapter 7). The Entities identified are probably right in the domain, but you'd want them to also handle their data isolate the data from other services, and replace the CRUD api with more domain oriented messages such as Upgrade customer status or Add address etc. for the customer entity.

8.4.4 Known Exceptions

Unlike the other anti-patterns, I can't think of any real exception where Same-old-way would be acceptable. Unless maybe you are yourself that ISV that knowingly brands its own product as something it isn't for "marketing reasons".

Kidding aside, as I said before, the main problem is actually noticing that you have a lot of friction in your development and that the reason is that your not actually implementing SOA. The operative word here is friction. SOA itself is neither a panacea nor an end-goal. If whatever architecture you are using is viable for your problem and serves you well use it. If, however, you've turned to SOA because the "Same old way" architecture was problematic, don't expect that repeating it using new tooling and technologies will solve the problem.

For instance, 3-Tiered architecture mentioned above, is a viable architecture. There are many successful deployments of 3-tiered based solutions. However 3-tiered architecture will not give you the flexibility that an SOA will give you no matter how you cut it. It is a different architecture with different pros and cons. As mentioned before, reintroducing it under a new name will probably not solve your problem.

"Same old way" is probably the most generic anti-pattern which can recur when we want to apply any new technology/architecture and struggle with what it means to actually implement it in the real world.

8.5 Summary

The anti-patterns chapter introduced us to some of the common pitfalls we are likely to make moving on to SOA.

- The Knot – introducing tightly-coupled services by point-to-point integration.
- Nano Services – cutting the services too small and getting an unmanageable soup of services
- Transactional integration – forgetting service boundaries and coupling services together
- Same old way – not fully understanding what SOA is and isn't and falling into using our previous architectures

We've mentioned at the beginning of this chapter, that the second part of the book takes a look at different aspects of SOA in the real world. After looking at anti-patterns. The next chapter will look at another aspect of real-world SOA, which is that real problem are big and complex so that a single pattern can't solve it. To that end, we will go over a case study of an end-to-end solution integrating several patterns together into a grater whole.

8.6 Further reading

Table 8.2 resources for further reading on topics covered in this chapter.

Topic	Resource name/link	Why
The Knot	Big Ball of Mud , Brian Foote, Joseph Yoder http://www.laputan.org/mud/	Explains when it is ok to have a mess
Transactional integration anti-pattern	The Byzantine Generals Problem, LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf	Basis of distributed consensus
Transactional Integration anti-pattern	Shootout at transaction coral; BTP VERSUS WS-Transactions – Roger Sessions http://www.objectwatch.com/newsletters/issue_41.htm	More discussion on why transactions between services are bad
Transactional Integration anti-pattern	Transactional Processing Cheat-sheet - Christophe Bare http://www.cbare.org/writing/Transactions/transactions.html	Thorough explanation of transactions

9

Putting it all together – a Case Study

SOA Patterns details a lot of different patterns. Each pattern handles just one aspect of building a solution like security, scalability, integration etc. However a real system has lots and lots of different challenges that need to be solved. It is interesting to see how different patterns can play together to provide a single cohesive solution – and that's what this chapter is about.

The chapter is divided into two parts: case study and solution. The case study presents the problem, describing a system that was developed using the service oriented architectural style and the quality attributes or architecturally significant requirements of that project. The second section, taking most of the chapter, presents the solution and the patterns that comprised it. The main idea behind this chapter is to demonstrate how putting multiple patterns to work together provides a larger whole. It also provides some highlights of the reflection of patterns into code on one hand and getting to patterns from requirements on the other hand to demonstrate how the patterns fits in the development lifecycle.

9.1 Case Study

In order to present a real-world solution that integrates several patterns in a meaningful way, we need a reasonably sized project. To that end I am going to present a system I worked on a few years ago where a lot of the patterns discussed in the book have been implemented and used. Incidentally that is also the system that got this book delayed by three years (working on it I hardly had time to breathe) so it seems fair it should at least serve as an example. We'll start by looking at the general characteristics of the system, followed by the some of the architectural requirements and the mapping to relevant patterns

9.1.1 The system

One interesting way to explore the world around us is through visual search. The idea is simple enough: see something of interest, take a picture of it with your mobile phone camera, send it to a service and get back relevant information. A public solution that performs this is Google Goggles, the system at hand did essentially the same with 2 key differences:

- The system supported multiple ways to send in images, namely, via video phone call, SMS, Email and Apps on various platforms (Goggles is only used via apps.)
- The system was an OEM white-label solution for content providers. i.e. it was a Software as a Service solution to provide visual search

Figure 9.1 below lists the main business services comprising the system. The system had four services for providing visual search (as mentioned above) – each a different business offering. It had a service for managing interactions where clients design the experience users would get when a search yielded a result; a service for managing advertising campaigns and a few more "classic" services like billing and reporting

Figure 9.1 Few of the business services the image search system exposes. The services include several ways to perform visual search (via email, app, etc.), Billing, interactions management (the result for a link)

etc.

SERVICE ORIENTATION VS. SERVICE ORIENTED ARCHITECTURE

Service Orientation vs. Service Oriented Architecture is a topic not in the scope of this book. It is, however important to discuss it in the context of a case study for two main reasons

- Service Orientation is a stepping stone towards deciding which services an SOA will include
- SOA services and business level services (i.e. services originating from Servicer Orientation analysis) are related but not necessarily have a one to one relationship

So, what is Service Orientation?

In a nut shell, Service Orientation is an approach for analyzing some of the aspects of Enterprise Architecture - specifically functional decomposition, business process and data architecture. Applying Service Orientation means focusing on breaking business capabilities and functions into business level services. The business level services are logical components whose composition and interaction provide the business's processes needed by the business.

Service Oriented Architecture, as explained in the first chapter, is an architectural style (i.e. a software concept) concerned with building inter-connected coarse grained components. SOA focuses on flexibility and composition (e.g. with the extra emphasis on the interface etc.). Indeed, the resemblance in name between SOA and "Service Orientation" is not incidental and, SOA is a good fit for implementing this approach.

In a sense the business services and business processes identified at the "Service Orientation" level are the requirements that are fed into the architecture, technology mapping and implementation at the software level where SOA plays.

Looking at the business services gives us a high-level overview of the functionality of the system and provides us hints toward partitioning the solution into services. When designing an SOA the next step would usually be to Understand and analyze the business processes in order to gain insights into what messages and contracts are needed. We'll see some of the results of such analysis (the analysis itself is not in the scope of the book) in the next section, however before that we also need to take a look at some of the system's quality attributes.

9.1.2 Quality Attributes

Another source of requirements that helps us understand how to design a solution are the architectural requirements – expressed as quality attributes scenarios. Quality attributes scenarios, as briefly mentioned in chapter 1 and demonstrated throughout the book serve as good way to identify candidate solutions that can help handle them. It is worthwhile mentioning that expressing quality attributes as scenarios carries many additional benefits, including achieving better understanding of the requirement, allowing building tests to demonstrate the quality, means to prioritize and evaluate an architecture etc. See the further reading section for some links on Quality Attributes

Table 9.1 below list some of the system's quality attributes scenarios along with the candidate patterns to handle the scenario

Table 9.1 Few of the case study's Quality attributes and the patterns that were used to tackle them

Quality attribute	Scenario	Relevant patterns
Adaptability/ changability (add/remove feature)	During development and operations, a change in a component will only affect the direct components (for development and production) Once in production, a change in an interface will be compatible at least one version back	Edge component (chapter 2)
Unplanned Downtime	Under normal conditions, a failure in a single component will not result in call termination.	Service Watchdog (chapter 3) Service Instance (chapter 2)
Time To repair/detect	Under normal conditions, the system will detect a failure in a component in less than 5 sec.	Service Monitor (chapter 4)
Deployment	Under normal conditions, the system will not require manual configuration to work. Under normal conditions, deploying a new version will l be	Inversion of communications, (chapter 5) Reservation (chapter 6)

	done by xcopy.	
Scalability	Under all conditions adding additional hardware units ("Deployment units) will enable linear growth in image database capacity.	Gridable service (chapter 3)
Cost	The cost of a deployment unit shall not exceed 1K$	Gridable service

Equipped with an understanding of the functionality and some of the quality attributes needed we can move on to the more interesting part – the solution

9.2 The Solution

The system that was constructed to handle the above mentioned requirements, evolved over time. Initially the system only had to handle identification in 3G video calls and small numbers of links then the business added requirements for SMS and eMail followed by a demand to handle large number of links and open the platform for mobile apps and general internet use. As you can probably guessed, we decided to build the system based on SOA principles. For one, that helped us meet the system's requirements. More importantly It helped constantly adapt the system to the changing requirements. SOA's flexibility and affordance for composition allowed us to add more components (services) as well as evolve the internal structure of existing services while keeping the system working.

This section will look at few of the SOA patterns that were put into use to make this happen. Let's start with a quick over view of the players (also depicted in figure 9.2 below) – some of the services that the system contained:

- Email Gateway, MMS Gateway & Phone client (green in figure 9.2)– Adaptor services that translated external protocols to internal ones (see more in section 9.2.1)
- SIP Listener, RTP Listener, Player, Call Recovery and 3rd Party GW (red)- services that handle different aspects of 3G video calls
- Billing, Interactions and Campaigns (black) – The B2B oriented services: bill different clients based on what users did with the system,(e.g. an SMS that was identified by the system and resulted in link to client would cost 0.45$, one that resulted in a purchase would cost 2$ etc.) setting interactions and advertising campaigns
- Liveliness Monitor, Monitor and Watchdog (lilac)– various "technical" services in charge of keeping the system going (see 9.2.3)
- Statistics collector, Reporting (cyan) – services responsible for collecting what's happened in the system and turning that into data for reports
- Identification and ID Workers (orange) – the image identification system (see 9.2.1)

Figure 9.2 Some of the services in the case study's system that implement the system's business services. For example providing identification by email requires choreography of the email gateway, identification (with its id workers), watchdog, statistics collector, monitor, and billing. The sections below will use these services in different contexts to demonstrate different patterns.

I mentioned that business services identified serve as requirements to the partitioning into services and that the mapping is not necessarily one to one. Table 9.2 below shows the mapping of the business services to the services that enable them. Section 1.2.2 on the use on Inversion of Communications will expand more on how a sample business service gets implemented by the interaction and composition of several services.

Table 9.2 mapping from business services to services that implement them. For example the MMS business service is supported by the MMS gateway, identification, watchdog, statistics collector and campaign services.

Business Service	**Services involved**
App Image search	Phone Client, Identification, Id Workers, Watchdog, Billing, statistics Collector, Campaigns

Email Image search	Email Gw, Identification, Id Workers, Watchdog, Billing, statistics Collector, Campaigns
MMS search	MMS GW, Identification, Id Workers, Watchdog, Billing, statistics Collector, Campaigns
3G Video call search	SIP Listener, RTP listener, Player, Call recovery, Identification, ID Workers, Watchdog, Billing, statistics Collector, Campaigns
Billing	Billing, Reporting
Campaign management	Campaign, Reporting
Interactions	Interactions, Reporting, Campaign

Ok, so now we have a "bunch of services" – but a bunch of services does not make a system. A system is made when components, or services in our case, cooperate and work together towards fulfilling a purpose The following sections will look at how SOA patterns help weave the different services into a system.

9.2.1 Structure (Edge Component, Gridable Service, Parallel Pipelines)

We'll start off by looking at the structure of the system and some of the patterns that we're applied there. The first pattern we'll take a look at is also the first pattern that appears in the book (way back at chapter 2). Edge component is a basic pattern and as an architectural pattern it can be applied in many levels. A quick reminder, Edge pattern is defined as follows:

> **Add Edge Component(s) to your service implementation to isolate business logic from contracts, protocols, end point technology and other cross-cutting concerns**

When working on the system we've implemented this pattern in multiple levels. First we can see this "in the large" or at the architectural level where the different protocols to perform a visual search (email, app, MMS and 3GVideos calls) were separated from the business logic performing the visual search itself. Conceptually it can be thought of as a single service performing visual search with the different gateways serving as Edge components – as visualized in Figure 9.3 below

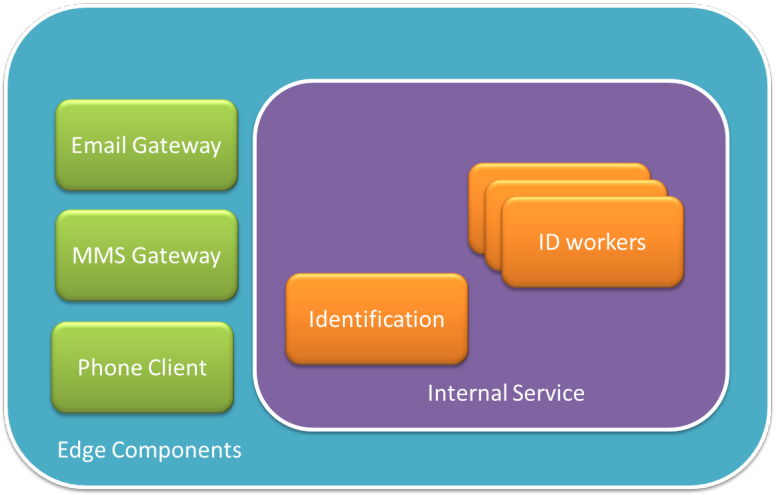


Figure 9.3 Application of the Edge component pattern at the architectural level. Edge services translate external protocols into internal protocols of the system. For example the email gateway works with IMAP protocols to get emails, extract the images from them and then calls the identification service to perform a search

CAN AN EMAIL GATEWAY BE CONSIDERED AS A SERVICE?

An interesting side discussion is whether the different gateways mentioned in this system can be considered services. We all know that SOAP web-services are services right? But email? MMS?

Well, first I'd like to say that using SOAP and web-services does not automatically mean you have an SOA and a service. For that matter you can take any object you have in the system wrap it with a SOAP enabling technology like WCF or JAX-WS and you'd just get a fancy way to do RPC.

A component is a service if it adheres to the definition of service. Let's take another look at the definition of SOA from chapter 1:

"Service Oriented Architecture (SOA) is an architectural style for building systems based on interacting orchestrating loosely coupled, coarse –grained, and autonomous components called services. Each service exposes processes and behavior through contracts, which are composed of messages at discoverable addresses called endpoints. Service's behavior is governed by policies which that can be set externally to the service itself. "

The different gateways do fall under this definition. We'll take the email gateway as an example

- Course grained and autonomous – The component lives on its own it can handle everything that is related to email and it communicates with other services to provide business functions. Note, for example, the image search service is not available the email gateway does not fail (it can still return an email reply that the system is unavailable) – it is the system as a whole that fails in this case to provide the business value

- Use of contract and messages at discoverable address– The contract is based on a known protocol –IMAP where the message structure is an email which must have an image attachment in one of supported formats, which is addressed to a specific mailbox (the endpoint)

- Governed by policies – The policies that can be set here are the origin of emails accepted, that origin email address are verifiable (DKIP/ IPS), that messages are signed or not etc.

To summarize this point – First off, no, I am not suggesting you write all your services to use email or replace your ESB with an Exchange server. However, services can come in different shapes and sized, using different protocols and technologies. What makes something a service is the way it is constructed and the way they interact and used within a system.

The system also implemented the Edge Component pattern "in the small" - at the code level. Code snippet 9.1 below shows a simple WCF data contract and operation contract, which define an external interface (something other services can consume/use) for sending MMS messages.

Listing 9.1 Simple WCF contract for sending an MMS message

```
    [ServiceContract]
    public interface IHandleSendMms  #1
    {
        [OperationContract]
        int SendMms(SendMmsRequest eventOccured);
    }

    [DataContract]
    public class SendMmsRequest : ImEvent #2
    {
        /// <summary>
        /// end user's number. should be in international format:
+[country-code]number. Example: +491737692260
        /// </summary>
        [DataMember]
```

```
        public string ToNumber { get; set; }
          /// <summary>
          /// service's number, usually a short-code. Example: 84343
          /// </summary>
          [DataMember]
          public string FromNumber { get; set; }
          /// <summary>
          /// Text, as byte array. Use Encoding classes to do it.
          /// </summary>
          [DataMember]
          public byte[] TextAsByteArray { get; set; }
          /// <summary>
          /// Image, as byte array. Can be: jpg, gif, png, bmp. (jpg rulez!!)
          /// </summary>
         [DataMember]
          public string ImageExtension { get; set; }
          /// <summary>
          /// the mms message should have a subject. just put something
there.
          /// </summary>
          [DataMember]
          public string Subject { get; set; }
}
```

#1 – Service Contract attribute is a way to tell WCF that this is would be a message in a contract
#2 – When defining complex types, Data contract attribute tells WCF what is signigicant for the message contract

The code snippet below (9.2) shows one of the methods of the service that fulfill the contract for sending out the MMS. We can see it translates the external message into internal structure. We can also see that it adds missing information which does not appear in the external contract. In this example the file extension (ie. The type) of the image that is sent via the MMS is needed internally. The edge component looks at the image and infers its type so that the business logic can concentrate on sending MMSs and not image parsing.

Listing 9.2 conversion of external message (contract) to an internal construct in an Edge Component. The code is taken from a class that implements the WCF contract (the "service" hosted by WCF)

```
public int SendMms(SendMmsRequest eventOccured)  #1
        {
            var eventContext = eventOccured.ToString();
            if (log.IsDebugEnabled)
                log.Debug("inside 'SendMms', event context = [" +
eventContext + "]");
           var fromNumber = eventOccured.FromNumber;
            var sender = mmsSenderFactory.Get(fromNumber);
            if (null == sender)
            {
                if (log.IsWarnEnabled)
                    log.Warn("cannot get mms sender derived from '" +
(fromNumber ?? "null") + "'");
                return 0;
            }
            IMmsSubmitResponse response;
            try
```

```
            {
                var extension =
GetImageExtension(eventOccured.ImageAsByteArray); #2
                var mmsMessageDetails = new
MmsMessageDetails(eventOccured.ToNumber,
                  eventOccured.TextAsByteArray,
                  eventOccured.ImageAsByteArray,
                  extension),
                                                eventOccured.Subject);   #3
                response = sender.Submit(mmsMessageDetails);   #4
            }
            catch (Exception ex)
            {
                log.Error("cannot send mms message, context = [" +
eventContext + "]", ex);
                return -1;
            }
            if (log.IsInfoEnabled)
            {
                var responseMessage = (null == response) ? "null" :
response.ToString();
                log.Info("sent mms with event context = [" + eventContext +
"], response = [" + responseMessage + "]");
            }
            return 0;
        }
```

#1 – The external message SendSMSRequest accepted from the wire
#2 – Adding missing information that doesn't appear in the external contract but needed by the internal one (finding out the file extension of the image attached)
#3 – Conversion into an internal structure MMSMessageDetails
#4 – Calling the internal service with the internal structure

Edge Component helped us build individual services. We also needed away to tie up services together to bring fulfill the business services. This was especially challenging for video calls. Performing visual search for video search is basically a clientless service -the client side application is the video call provider which creates a two-way video channel as well as passes DTMF tones (key strokes). This means that a lot of the handling of video calls requires stateful services like handling the incoming video stream, generating the outgoing video stream and maintaining the state of the client

To solve this problem we've applied the parallel pipelines pattern. The parallel pipeline pattern, as described in chapter 3 is:

> **Implement the Parallel Pipelines pattern, where you break the process into sub-tasks, add a queue between them, and make each sub-task an independent component.**

Figure 9.4 below shows an excerpt from the process of handling videos calls and demonstrate how the parallel pipelines pattern had been put to use. Each service implement just one sub-task related to handling the video call visual search, and the work is handed over from service to service to complete the overall business process. Specifically, the RTP

listener accepts an incoming RTP (real-time-protocol) of video in H263 format and keystrokes from the users. Depending on what was decoded (image or keystroke) it sends it over to either the identification service – which performs the search or call-flow that decides what needs to be shown to the user either instructions on what to do next (point the phone's camera at something interesting) or the result (video / get a coupon/ get a hyperlink etc.) and the player takes care of actually showing the user what they need based on the decisions from the call-flow.

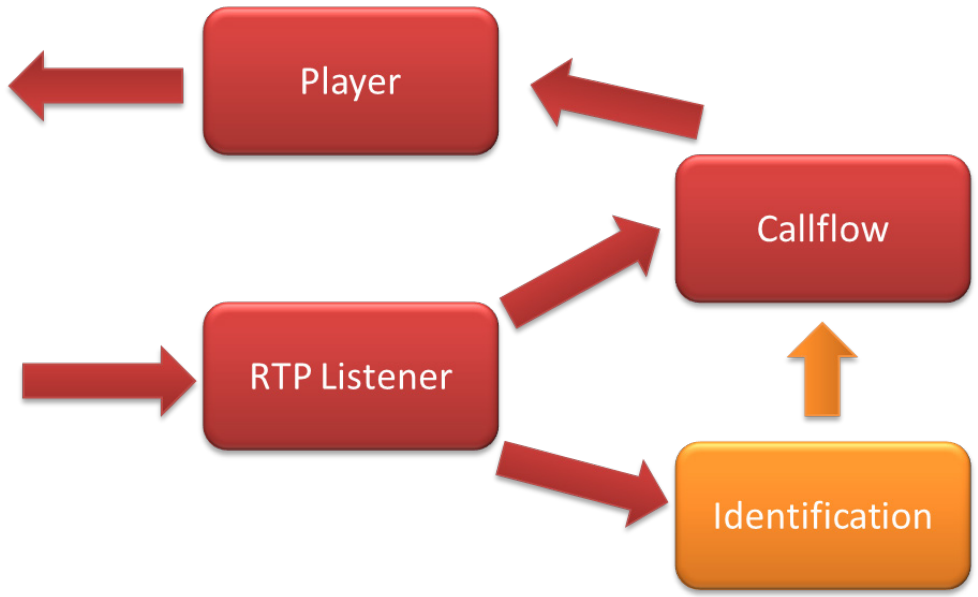


Figure 9.4 an implementation of a parallel pipelines pattern. The RTP listener takes the sub task of decoding RTP, sends the results to Call-flow and Identification, Identification takes the subtask of performing the visual search, call-flow takes the subtask of understating what happening and deciding what the user needs to see (instructions, the results of the search etc.) and the player takes the subtask of providing video to the end-user. All together they perform the business service of visual search over videocall

Using Edge Components and parallel pipelines helped make the system tick – but to solve the core problem of the system – searching based on images, we needed another pattern the Gridable Service pattern (see chapter 3)

Introduce grid technology and concepts to the service, with the Gridable Service pattern, to handle computationally intense tasks.

You'd have to trust me on that but visual search is indeed a "computationally intense task" which in a nut shell involves building an identifier or signature to each image in the database (something done offline) and then when receiving a new image building a signature for that image and then comparing the results. You can also disqualify some of the results based on different aspects of the image for example application icons have different features from general images. On top of that you can use all sorts of meta-data to narrow the search for example if you have a client with a magazines in Germany and another with a newspaper in the US you can decide not to check one of the databases based on the location the image was taken in. Naturally in practice all of this is rather complicated and involves math that beyond the scope of the book and me to be honest.

I do know a bit more about software architecture though, so let's focus on that. Figure 9.5 below illustrate how the Gridable Service pattern was utilized in the system. The identification service is the grid root and it distributes a search for different machines that register on the grid. Within each machine there's a local database that contains part or a shard of the overall database (we used Cassandra for that) and computation agents that work against it to perform the actual image search

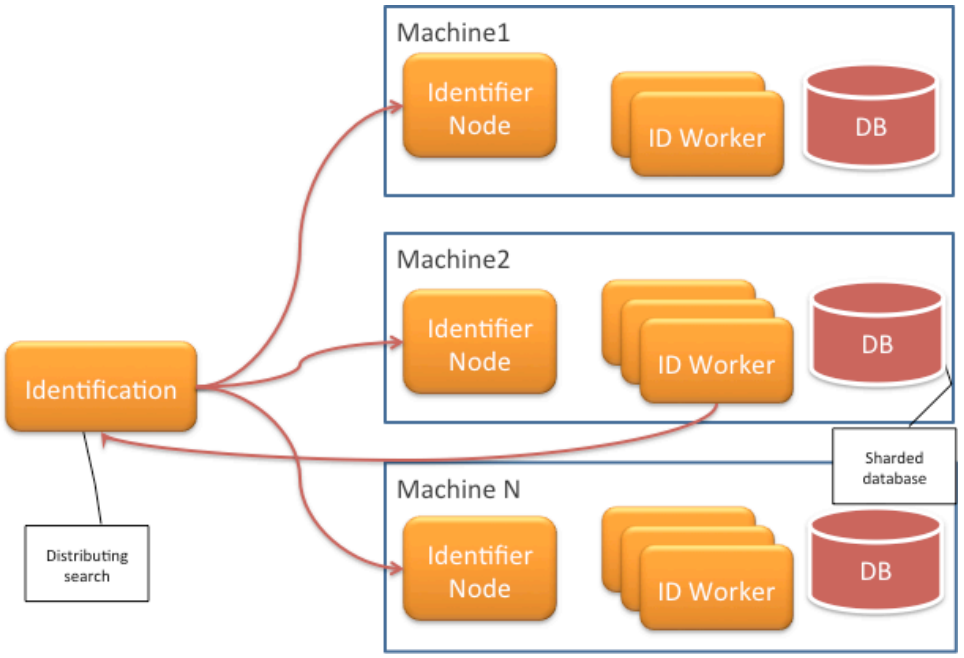


Figure 9.5 Gridable service pattern applied. The identification service is the grid's root distributing the search to various machines. Each machine has a local manager that employs ID workers which are computation engines hitting a small(er) part of the image database

I guess one can argue it was more of a computation cluster than a grid but we did have some grid traits in the sense that nodes id come and go dynamically (first on failure and growth scenario and later elastically as we we're reading the system for the cloud). Thus, one of the roles of the root node (the identification service) is to periodically check if new servers joined the gird (crashed/failed servers are removed when a call to them fails). We can see the code for that in snippet 9.3 below. The identifier accepts is thread of execution from Retlang – a .net open source library for Erlang style concurrency and uses it to schedule a timed event to recheck for new servers availability within the grid.

Listing 9.3 a class to check for exsistance of new workers (handlers in the code) that have joined the grid. The refreshMarchersAction does the actual discovery of new workers and it uses information received from the Service watchdog (see section 1.2.3 below)

```
    public class HandlersRefresher
    {

        private readonly IFiber dispatcherFiber;
        private const int HandlersUninitializedValue = -1;
        private int idealNumberOfHandlers = HandlersUninitializedValue;
        private static readonly ILog log =
LogManager.GetLogger(MethodBase.GetCurrentMethod().DeclaringType);
        public const int RefreshIntervalInMs = 30*1000;
        public const int TimeToFirstCheckInMs = 10*1000;
        private readonly Action<bool> refreshMatchersAction;
        private ITimerControl timerControl = null;

        public HandlersRefresher(IFiber dispatcherFiber, Action<bool>
refreshMatchersAction) #1
        {
            if (dispatcherFiber == null)
                throw new ArgumentNullException("dispatcherFiber");

            if (refreshMatchersAction == null)
                throw new ArgumentNullException("refreshMatchersAction");

            this.dispatcherFiber = dispatcherFiber;
            this.refreshMatchersAction = refreshMatchersAction;
        }

        public int CurrentIdealNumberOfHandlers
        {
            get { return this.idealNumberOfHandlers; }
        }

        public void InspectHandlers(IFrameHandler[] handlers)
        {
            var numberOfHandlers = handlers == null ? 0 : handlers.Length;

            ScheduleRefreshIfNotAlreadyScheduled();

            if (numberOfHandlers > 0 && numberOfHandlers >=
idealNumberOfHandlers)
```

```
            {
log.DebugFormat("Setting the ideal number of handlers to {0}",
numberOfHandlers);
                idealNumberOfHandlers = numberOfHandlers;
            }
        }

        private void CancelRefreshSchedulingIfExists()
        {
            if (timerControl == null)
                return;

            log.Debug("Stopping scheduled handlers refresh");

            timerControl.Cancel();
            timerControl = null;
        }

        private void ScheduleRefreshIfNotAlreadyScheduled() #2
        {
            if (timerControl != null)
                return;

            log.DebugFormat("Scheduling handlers refresh every {0} ms.",
RefreshIntervalInMs);
            timerControl = dispatcherFiber.ScheduleOnInterval(() =>
refreshMatchersAction(false), TimeToFirstCheckInMs, RefreshIntervalInMs);
        }

    }
}
```

#1 Get an execution thread and the a method that can check for new workers
#2 schedule a new check event

If we want to understand how the refreshMatchersAction did its magic as well as how messages travelled between all the services we need to take a deeper look at the communications mechanisms in the system and the patterns they've used.

9.2.2 Communications (Inversion of Communications, ServiceBus, Saga, Reservation)

At the heart of the communications mechanism of the system lie software component called eventBroker. As implied by the name it implemented the inversion of communications pattern (see chapter 5):

> **Implement the inversion of communications pattern by supplementing SOA with Event Driven Architecture (EDA) , i.e. Allow services to publish events streams of changes that occur in them instead of calling other services explicitly .**

The system use of event semantics extended both to events notifying something happened e.g. Frame-Arrived event when a new image was made available for identification in a video call as well as for asynchronous request e.g. Send-Coupon Request sent from the

call-flow. Code snippet 9.4 below shows how the eventBroker is used from the consumer side. The SendCoupon method prepares a request with all the needed data and then calls the eventBroker's RaiseEvent to dispatch this to an unknown handler – another service, that would send the actual sms with the coupon inside.

Listing 9.4 sample code from the callflow service showing rasing an event.

```
        public void SendCoupon(string title, String targetUri)
            {
                var recepient = callerPhoneNumber.Value;
                Logger.DebugFormat("sending sms to '{0}', title is '{1}',
    content is '{2}'", recepient, title, targetUri);
                SendCouponRequest eventDetails = new SendCouponRequest()
                                                        {
                                                            Recepient =
    recepient,
                                                            Content = new
    Uri(targetUri),
                                                            Title = title
                                                        };  #1
                eventBroker.RaiseEvent<SendCouponRequest>(eventDetails); #2
            }
```

#1 preparing an event, the code decides it needs to send out a coupon so it creates a coupon request
#2 the next step is to put the event on the bus in the hope that someone would take care of it (at least eventually) – if the code needs confirmation it would have to listen on a CouponSentEvent that some other service would publish

The second part of the name eventBroker, unlike the first part, is actually misleading for the eventBroker was actually a bus (agent on each server and not a centralized node) and it implemented the ServiceBus pattern (see chapter 6):

Implement the Service Bus pattern and use a unified messaging infrastructure, for message transformation, mediation, routing and invocation infrastructure

The eventBroker indeed isolated the business logic from any information related to the target or targets of the event (even though the actual messages were sent over http using WCF). Sometimes an event was sent to multiple subscribers, sometimes it was sent to different subscribers based on context (more on that later), sometime the event was broadcasted etc. The eventBroker also encapsulated failures e.g. retries, ignoring cyclic messages if there was a timeout, skipping failed recipients if they were optional etc.

We can see another interesting capability of the eventBroker in code snippet 9.5 below. The code shows another method from the callflow service that handles sending a request to play a movie to a video call end-user. Overall the method is very similar to the SendCoupon method mentioned in snippet 9.4 above. The main difference lies in the last row of the code "eventBroker.RaiseSagaEvent(playMovReq);"

Listing 9.4 sample code from the callflow service showing rasing a saga event.

```
public void PlayMovie(String mediaLocation, bool loop, string
interactionID)
        {
            var playMovReq = new PlayMovieRequest(SessionId, mediaLocation,
loop, interactionID);
            if (Logger.IsDebugEnabled)
                Logger.Debug("in saga [" + playMovReq.SessionID + "] about
to play movie '" + mediaLocation.ToString() + "', loop = " +
loop.ToString() + " interaction ID = '" + interactionID + "'");
            eventBroker.RaiseSagaEvent(playMovReq);  #1
        }
```

#1 Some events have a shared context with previous and future events, in this cases they can be raised inside of a Saga

While the event in snippet 9.4 is not related to other events and is completely standalone, the event in snippet 9.5 is raised as part of a Saga. i.e. the event is sent as part of a series of related events which are a part of a single business process. Here's a reminder for the definition of Saga as it appears in chapter 5:

> **Implement the Saga pattern and break the services interaction i.e. the business process, to multiple smaller related business actions and counteractions. Coordinate the conversation and manage it based on messages and timeouts.**

Why did we need Sagas? One reason is that we also used the Service Instance pattern (more on this later) another is failure scenarios for instance you don't want to bill a client if the end user did not receive a result that that warrants that billing. When there's a problem with one of the services involved in the saga (whether it is a communications problem or a business logic one) it can throw a Saga Fault event and some other service or services will try to handle/recover from the failure. Let's revisit the parallel pipelines scenario we discussed above to demonstrate this. In figure 9.6 below we see the RTL listener raising an event that needs to go to the call flow (1) which in turn fails. Since this is a communications problem the eventBroker will raise the Saga-Fault event – without the business logic intervention which the Call recovery service listens to. The call recovery will then try to handle the failure e.g. allocate another call flow instance to the saga so calls can be completed

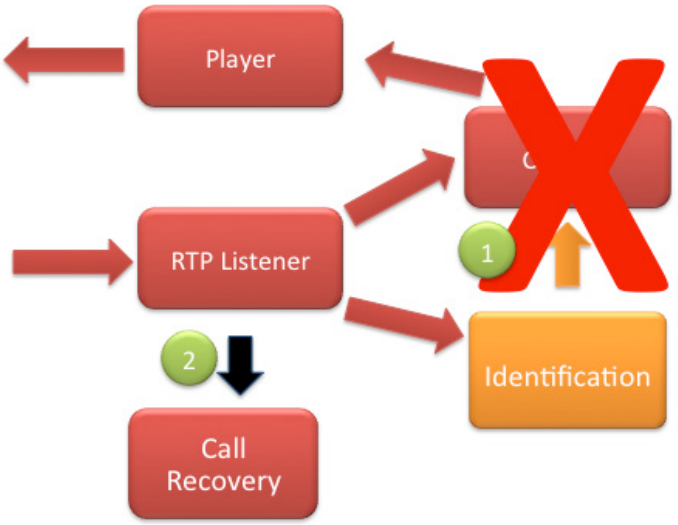


Figure 9.6 Illustration of a case where the eventBroker raises a Saga Fault event. The eventBroker in the RTPListener tries to send an event to the Callflow (1). When it fails it raises a Saga Fault event and another service, the Call Recovery, listen to it and tries to contain the failure.

The initiator or a saga also set the context for that saga and another role fulfilled by the bus (the eventBroker) was then to route the events based on that context.To illustrate this think of a simple scenario: All the different gateway services : Email, MMS and Apps send out the same event about having a new image ready for identification. The result of the search needs to go back to the right gateway. The identification service will still emit an Identification-Found event regardless of where the request originated. We don't want the service to know about the recipient as that the whole point of having a bus. We don't want to broadcast the result because it is both not scalable and creates needless load. Instead, as mentioned we employed context based routing.

Each service contract included a list of the events that the service listens to as well as a list of contexts where these messages are applicable. Code snippet 9.5 below, shows how the list of contexts – the different Participate attributes (annotations in Java speak) that the Identification service listened to. In this case these are the different ways to perform a visual search.

Listing 9.5 – Sample contract listing the events that serice deals with as well as the contexts where these messages are relevant.

```
[ServiceContract]
[BroadcastStatus]
[Participate(Contexts.3rdParty)] #1
[Participate(Contexts.VidCall)]
[Participate(Contexts.Client)]
[Participate(Contexts.Mms)]
[Participate(Contexts.Email)]
public interface ImIdentifier : ImContract,
                                IHandleNewImageInSequence, #2
                                IHandleCallStarted,
                                IHandleCallEnded,
                                IHandleCallAborted,
                                IHandleSearchStarted,
                                IHandleInteractionStarted,
                                IHandleReadyForSearch,
                                IHandleImageIdentification,
                                IHandleReshardingOccured

{}
```

#1 – Participate attributes designate the contexts for which the contract is relevant
#2 – IhandleX is a generated Interface for an event, in this case the NewImageInSequence event

When a new event is raised, the serviceBroker, needs to find what types of services are subscribed and notify them as well as the notify other member of the saga (since each holds its own instance of the eventBroker). As some of the services had limited capacity and needed to be verified as participants we've also implemented the Reservation patter (see chapter 6):

Implement the Reservation pattern and have the services provide a level of guarantee on internal resources for a limited time

The eventBroker basically tries to connect to, what it thinks is, a free service and retries until it secures all the services it needs (or maxes out on retires). Code excerpt 9.6 below show the code for handling the reservation in the eventBroker. It basically sets a timeout for all the reservation to be done and then tries to reserve each of the candidates by the timeout. If some of the services deny the reservation the process tries to find new candidates. Once all the needed services are reserved it tries to "commit" the reservation with all the services. Note that services can also fail during a commit as well as after the saga is in use – The Saga Fault event mentioned above handles these situations.

Listing 9.6 code used to Reserve instances when a new member or members need to be added to the saga

```
        private IEnumerable<Uri> Reserve(IEnumerable<ProxyWrapper>
wrappers)
        {
            var timeToComplete = DateTimeOffset.Now + TIMEOUT;
```

```
            var uris=Reserve(wrappers, MAX_RETRIES,new
List<Uri>(),timeToComplete); #1
            return FinalizeReservation(uris) #2
        }

        private IEnumerable<Uri> Reserve(IEnumerable<ProxyWrapper>
wrappers,int retries,ICollection<Uri> failedUris,DateTimeOffset
timeToComplete)
        {
            var success = true;
var newUris = GetCandidateUris(wrappers, failedUris); #3
            if (null == newUris) return null;
            if (newUris.Count == 0) return newUris;
            failedUris = TryReserveNewMembers( newUris); #4
            if (failedUris.Count>0)
                success = false;
            if (DateTimeOffset.Now>timeToComplete)
                success = false;
            if (!success && retries==0)
                return null;
            if (!success && retries > 0)
                return Reserve(wrappers, retries - 1,failedUris,
timeToComplete); #5
            return newUris;
        }

        private ICollection<Uri> TryReserveNewMembers( IEnumerable<Uri>
newUris)
        {
            var notifier = new SagaNotifier(Id, Route, newUris,
OptionalMembers, LocalUri, Allocator);
            var failedUris = notifier.ReserveAll();
            return failedUris;
        }
```

#1 Initiating the reservation with given number of retries and an initial list of candidate URIs
#2 if we got all the reservations finalize it so all the services will be in the saga
#3 Get a list of URIs of services to reserve based on known services and the failed services thus far
#4 Try to reserve the services and get a list of failed reservations
#5 if we still have time and retries attempt to spare – retry to reserve the failed Uris

The proxywrappers are classes that encapsulate the communications to specific services (one wrapper per service) In order for the eventBroker to be able to allocate services to a saga it had to know where to find these services so it can create valid proxy-wrappers for them. Or in other words the eventBroker needs to know which and where the "live" or active service end points are. To understand how this was done we need to take a look at a couple of other patterns implemented in the system the Service watchdog and Service Instance.

9.2.3 Availability (Service Instance, Service watchdog , Monitor)

While most of the services like billing, gateways etc. were stateless, some of the services weren't especially within the different services that provided the visual search over 3G video calls. In that setup when a user calls the system she doesn't install any client – which means that the system has to maintain state on behalf of the user, it also needs to

constantly stream relevant videos to the user (instructions, results of search etc.) etc. We chose to implement these using the Service Instance pattern (chapter 3):

Implement the Service Instance pattern by deploying multiple instances of the service business logic.

The main reason for this choice was availability and failure handling concerns. Figure 9.7 below illustrates the problem solved by using Service Instance. Video calls were carried on E1 lines each E1 can handle up to 30 concurrent calls. Let's say we have one such line and it is connected to a single Call flow service. If you remember the service in stateful (hydrating/dehydrating state takes too much time) so a failure in one call may cause the whole call flow service to fail and take with it all the 30 calls resulting in dissatisfied customers and loss of business. Instead using a Call flow per caller you get better isolation – a failure only affect one caller. Incidentally it also made the service much simpler to program as there were less multi-threading and multi-tenancy issues to worry about.

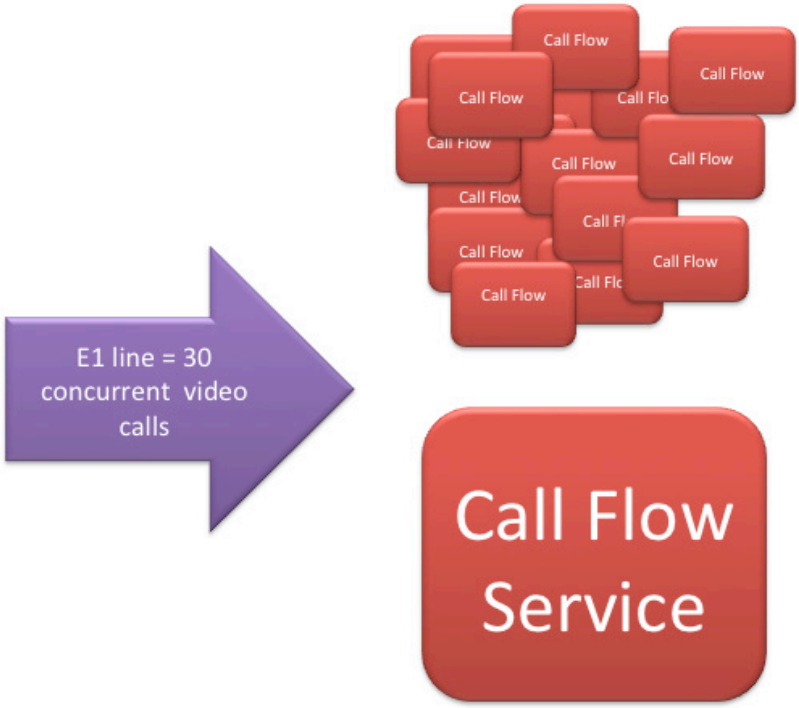


Figure 9.7 Using Service Instance pattern to minimize the effect of failure. An E1 line can transfer 30 concurrent video calls. When you connect them to a single call flow service, a problem with one may have the effect of bringing down the whole service cutting off 30 users from the system. If each callflow service only handles one call at a time. A failure will only affect a single caller.

Another place we implemented the service instance, which I already mentioned, is for the individual computation instances (ID Workers) as part of the identification grid (see figure 9.5 in the previous section). This was done both to have better isolation in cases of failure as well as to bring computation closer to the data for greater speed.

We've already talked about two component that were in use to handle failures, one was the Saga Fault event that services can raise when something is wrong, the other is the Service Instance pattern discussed above. The third measure against service failure that we're going to discuss is the implementation of the Service Watchdog pattern in the system.

The Service Watchdog pattern is defined (in Chapter 3) as follows:

> **Implement the Service Watchdog pattern, where a service actively monitors the state services, acts on potential trouble and tries to heal itself, as well as continuously publishes its status.**

Let's take another look at the failure scenario we've discussed earlier in regard to the Saga Fault event (see figure 9.8 below) The RTP listener service raises a saga event that supposed to get to the Call Flow service which it can't raise. The RTP will now try to raise a Saga Fault event but how do we know what the problem is? Here are few options:

- The Call Flow service is down
- The RTP Listner (acutally the eventBroker within it in this case) is failing somehow
- The network is down
- The whole computer which host the Call Flow service (and most likely other services) is down
- A combination of the problems above

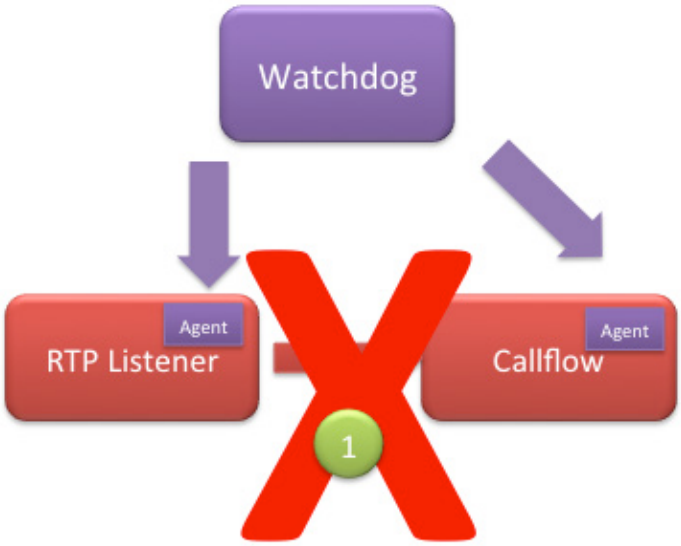


Figure 9.8 Illustration of the advantage of having a Watchdog. The RTP listener can't raise an event to the Call Flow. An external observer (the watchdog) enables figuring out if the problem is with the publisher, the subscriber, the network or any combination of them

The Service Watchdog, as an external observer is able to discern what the local (on the machine it is running) and expose that state to other components e.g. a Monitor so they can get the overall picture. In order for the watchdog to understand what is happening it used small agents that ran inside each Service. Code excerpt 9.7 below shows the watchdog's class that managed the agents (there was one instance per agent). The Agent proxy, created a named pipe channel to the service through which it periodically asked the health of the service and provided a liveliness report of other services. It also used the channel to ask services to shutdown in case of an orderly shutdown of the logical machine.

Listing 9.8 The proxy class used in the watchdog to manage its agents running within the services.

```
    /// <summary>
    /// Used on the host's side to manage the connection from the remote
proxy to the client
    /// </summary>
    public class WatchedServiceAgentProxy : IWatchedServiceAgent,
IDisposable
    {
```

```
        internal readonly Uri agentAddress;
        private IWatchedServiceAgent agentProxy = null;
        private int failures = 0;
        private static readonly ILog log =
LogManager.GetLogger(MethodBase.GetCurrentMethod().DeclaringType);
        private const int MAX_FAILURES = 3;
        private readonly ResourceContractInfo resourceContractInfo;

        public WatchedServiceAgentProxy(ResourceContractInfo
resourceContractInfo, int instanceIdentifier)
        {
            this.resourceContractInfo = resourceContractInfo;
            agentAddress =
AgentAddressProvider.GetAddress(resourceContractInfo, instanceIdentifier);
        }
     public Uri Address
        {
            get { return agentAddress; }
        }

        public void Dispose()
        {
            if(this.agentProxy != null)
                ((ICommunicationObject)agentProxy).Abort();
        }

        void EnsureAgentProxy()   #1
        {
            if (agentProxy == null)
                agentProxy =
ChannelFactory<IWatchedServiceAgent>.CreateChannel(
                    new NetNamedPipeBinding(),
                    new EndpointAddress(agentAddress));
        }

        void RenewFaultedAgentProxy()
        {
            if(agentProxy != null)
                ((ICommunicationObject)agentProxy).Abort();

            agentProxy = null;
            EnsureAgentProxy();
        }

        public LivelinessResult IsAlive()  #2
        {
            try
            {
                EnsureAgentProxy();
                var isAlive = agentProxy.IsAlive();
                return isAlive;
            }
            catch(CommunicationException ex)
            {
                throw new ServiceCommunicationException("Call to IsAlive
failed", ex);
```

```
            }
        }

        public string GetName()
        {
            EnsureAgentProxy();
            return agentProxy.GetName();
        }

        public void Shutdown()  #3
        {
            EnsureAgentProxy();
            agentProxy.Shutdown();
        }

        public void AcceptResourcesStatusBroadcast(ServiceStatus[]
resourcesStatus) #4
        {
            try
            {
                EnsureAgentProxy();
                agentProxy.AcceptResourcesStatusBroadcast(resourcesStatus);
                Interlocked.Exchange(ref failures, 0);
            }
            catch(CommunicationException ex)
            {
                if (resourceContractInfo.IsOptional)
                {
                    log.Info(string.Format("optional resource '{0}' cannot
be reached, ignored", resourceContractInfo.ContractName));
                    return;
                }
                if (failures > MAX_FAILURES)
                {
                    log.WarnFormat("Could not reach the watch dog agent on
pipe '{0}' for {1} times in a row. Renewing the agent proxy, exeption={2}",
agentAddress, MAX_FAILURES + 1,ex);
                    RenewFaultedAgentProxy();
                    Interlocked.Exchange(ref failures, 0);
                }
                Interlocked.Increment(ref failures);
            }
        }

    }
```

#1 Create a named pipe channel to communicate with the agent – Each logical server had a watchdog to communicate with locally installed services
#2 The watchdog periodically checked the liveliness of the service by calling the proxy's isAlive - which in turn asked the agent for the service's heath.
#3 On an orderly shutdown of the machine the watchdog requested the services to exist gracefully
#4 Every once in a while the watchdog updated the services with its current view of the world i.e. which services are up and where. This information was used internally by the eventBroker to manage sagas and locate event subscribers

We deployed one instance of watchdog per logical (virtualized or real) server. The Service watchdogs also formed a network between them and discovered new servers waking up (as well as going down). This enabled using the Watchdog as a poor man's Service Registry in the sense that it provided Endpoint management capabilities - a way to discover and locate available service endpoint across the whole system and some reporting capabilities (It was far from a real Service Registry because it lacked governance capabilities like managing versioning, SLAs, assets etc.)

We've seen that in order to achieve a desirable architectural capability we need to use several patterns together. This is of course also true for a whole system.

9.3 Summary

Let's take a quick look at all the patterns we've seen in use in the case study

- Edge component – separate business logic from technical concerns
- Parallel Pipeline – coordinate an effort by specialization (increasing throughput)
- Gridable Service – solving a computational intensive problem
- Inversion of Communications – adding flexibility
- ServiceBus – providing location transparency and communications
- Saga – tying together related events
- Reservation – securing instances to a saga
- Service Instance – breaking a service to multiple instances to increase overall availability
- Service Watchdog – monitoring health of local resources and reporting to a central monitor

When we built the system we also used quite a few other patterns such as:

- Aggregated reporting (chapter 7) building the reports by listening to the events already flying in the system
- Active Service (chapter 2) – having some services with their own thread of control (not just reacting to requests)
- Service Host (chapter 2) – shared code to insure services have some standard facilities (e.g. the watchdog agent)

However, as mentioned at the beginning, the point of this chapter is not to show-off how many patterns I know or implemented, rather there are three main goals for writing this chapter:

First, I wanted to provide a glimpse into implementation details of patterns. The patterns in the book are architectural, and as such they can have a lot of interpretations. The

technology mapping section of each pattern only touched on implementation briefly, and here we saw a little more detail of what's involved.

The second reason is to demonstrate a move from requirements to services. Naturally this is just touching the process but I hope that it provided some insight into what's involved. I think it is also important to show the difference and relation between business services and the architecture that is built to support them

The last goal for the chapter is to demonstrate how using multiple services together increases their overall usefulness and provide means to create a system. The system illustrated, here is just one way to compose the services together. Other requirements will utilize a different set of patterns with a different set of relations and ultimately different architectures and designs. The important point is that patterns can work together to provide a cohesive whole.

9.4 Bibliography

http://arnon.me/2010/05/quality-attributes-introduction/
http://arnon.me/2010/05/utility-trees-hatching-quality-attributes/
http://www.sei.cmu.edu/reports/03tn012.pdf

10
SOA vs. the world

The second part of the book covers different aspects of implementing SOA. We've started with anti-patterns that discuss some of the things that can go wrong. The next chapter presented a case study of how different patterns interact and complement each other. This chapter takes a look at the impact of other architecture styles and trends on SOA. We're going to cover

- **REST:** What's the relation of REST and SOA - friends? Foes? Can they work together?
- **Cloud:** Is SOA a good fit for cloud-based deployments? How does the cloud affect SOA
- B**ig data:** NoSQL is starting to mature with offering from the big vendors both in the advanced analytics front (e.g IBM and EMC offering a distro of Hadoop) and the big data in real-time (e.g. IBM Streams and SAP HANA) - how does SOA fit in?

Let's start by looking at the REST architectural style, which many see as an alternative to SOA.

10.1 REST vs. SOA

In recent years the REST architectural style has become very popular with a lot of companies (esp. internet ones like twitter and facebook) building RESTful APIs and a lot of other companies building value-add services, called mashups, by using these APIs.
Wikipedia defined mashups as follows:

> In *Web development*, a **mashup** is a *Web page* or application that uses and combines data, presentation or functionality from two or more sources to create new services. The term implies easy, fast integration, frequently using *open APIs* and data sources to produce enriched results that were not necessarily the original reason for producing the raw source data.

> The main characteristics of the mashup are combination, visualization, and aggregation. It is important to make existing data more useful, moreover for personal and professional use. (Wikipedia mashups http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)

Hey, wait a minute, this mashup thingy sounds a little like SOA so to help clarify things I'll explain the differences between REST and SOA and what's a RESTful SOA, but before that, let's take a quick look at what exactly REST is.

10.1.1 So, What's REST anyway?

REST which is short for REpresentational State Transfer, is an architectural style defined by Roy T. Fielding back in 2000 to describe the architectural style of the Web. REST basic component is the resource, which is addressable at an endpoint called URI. Fig 10.1 below illustrates the constraints that make REST style sets.

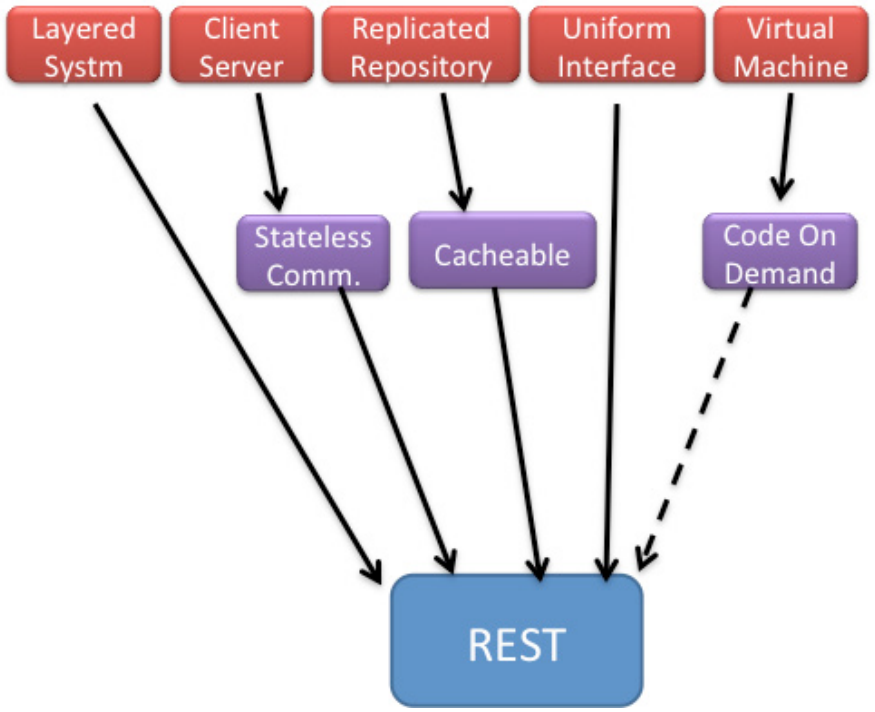


Fig. 10.1: The REST architectural style is derived from five base styles. Layered System, Client/Server, Replicated Repository, Uniform interface, Virtual machine

Let's look at the constraints one by one

- Layered System: The layered architectural style defines a hierarchy of components (layers) so that each layer can only know one level down. This promotes simplicity and the ability to enhance capabilities by adding middle-layers (e.g. firewall for adding security)
- Client/Server: introduces separation of concerns between consumers and the providers.
- Stateless communications: This constraint means that each request made from the client to the server should have enough context (state) to enable the server to figure out what to do with it. This is why, for example we have cookies that carry the session state from browsers to server.
- Replicated Repository: The idea behind this constraint is that it is ok to have more than one process provide the same service in order to achieve scalability and availability of data.
- Cacheable – cacheable constraint is the application of the Replicated repository constraint to the message level. Being cacheable means that messages can specify whether it is ok to cache them and for how long. This helps save on server round-trips, improve performance and decrease server loads.
- Uniform interface –_probably the most distinct characteristic of REST is the use of a limited vocabulary. HTTP, the most prevalent REST implementation, offers just 8 methods (GET, POST,PUT, DELETE and the lesser known, OPTIONS, HEAD, TRACE and CONNECT). The uniform interface makes it relatively easy to integrate with RESTful services. It also has a lot of impact on how you model them.
- Virtual machine: Or the interpreter, i.e. the ability to run scripted code- that's the base for code-on-demand (see below)
- Code on demand: An optional constraint that allows downloading code to client for execution (that's the javascript thingy that runs in our browsers.). Code on demand makes integration easier as clients can get code to handle the data they need instead of figuring out what to do with it

Another important aspect of REST is the use of hyper-media as the engine of application state (HATEOAS). HATEOAS means that replies from a REST service should provide links (URIs) as to the available options, based on the server's state, for moving forward from the current point. For example if a request to place an order was made, the reply can contain a URL for tracking the order, a URL for canceling the order and a URL for paying for it etc. HATEOAS is an outcome of using a uniform interface– and just like chalk chase provides a map of the way to fulfill business goals when working with REST.

Ok so that's a view from 50,000 feet of REST, even so, we can see some similarities and differences to/from SOA

10.1.2 How REST and SOA are different

REST shares a couple of its constraints and components with SOA. Like with REST, Client/Server is also a basic building block of SOA as is the notion of layered system. On the other hand constraints like Uniform interface and Virtual Machine have are very foreign to SOA. We can see the whole picture in Figure 10.2 below, which illustrates SOA's influences versus REST's.

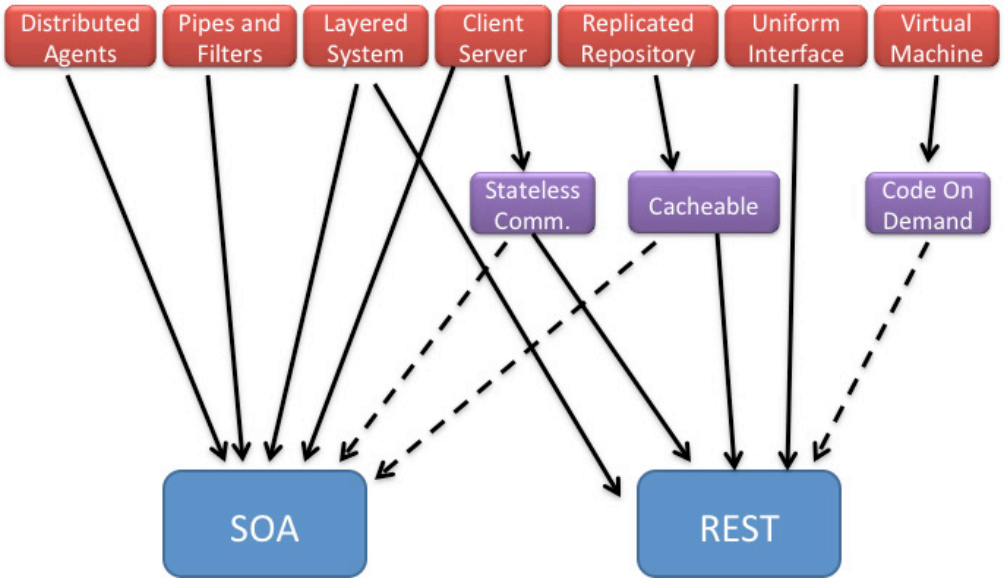


Figure 10.2: REST and SOA architectural constraints compared.

In addition to the Layered system style and client/server we can see additional two constraints that are optional in SOA. One of the optional constraints in SOA is the cacheable style. In chapter 5 we've talked about message exchange patterns and the benefits of sending immutable (e.g. versioned) data in messages. Immutable messages is SOA's way to specify cacheable messages, though explicitly specifying cacheabilty like in REST, is also an option.

The service instance pattern from chapter 3 is supportive of the Replicated repository constraint. Similarly, while stateless communication is not a must in SOA it is highly recommended (again see discussion in chapter 5).

SOA's benefits over REST include governance and planned reuse as well as high security standards and wealth of supporting components and message patterns (e.g. publish/subscribe). REST's advantages (especially REST over HTTP) include the ubiquity of the browser as well as serendipity of reuse.

As mentioned above, Virtual machine is very foreign for SOA, fortunately it and its derived constraint (code on demand) are optional for REST, which means we can combine REST and SOA to enhance SOAs reuse with REST reuse serendipity.

10.1.3 RESTful SOA

I find that RESTful SOA is beneficial when you want of have a dual API, in most other cases it is usually better to choose either SOA or REST (based on your specific needs) and stick with it.

How can we enrich SOA with REST? There are basically two approaches,

- Build a RESTful service and extend it to a SOA one
- Take a SOA service and extend it to a RESTful one

I would recommend the latter approach because SOA offers more flexible ways to connect services and has better tooling support. Also most chances are that in enterprise environment SOA related APIs would be more prevalent. That said in many cases you'd want to add REST to allow 3rd party integration as well as allow mobile clients to interact and consume services directly (sort of like the composite front-end pattern in chapter 6)

> **NOTE:** The Edge component pattern (see chapter 2) is a good approach to add a REST api on top of an existing SOA, you can even use technologies like Apache camel which enable flexible routing from external interfaces to internal ones.

The REST API and the SOA one will look radically different. REST comes with a hierarchical noun oriented API with SOA having a shallow verb oriented *API (both for event orie*nted and web-services oriented apis). However, I find that mapping between the two is more straightforward than it may seem.

> **A COUPLE OF THINGS TO REMEMBER:**
>
> - Different resources can map to a single service. For example if you have an orders and products resources. The order resource may have a GET /orders/<order id> URI to see orders details, and a GET /products/orders/ to see the different orders a product participates in – both might be mapped to an order service with 2 messages in its contract such as ListOrderDetails and GetProductOrders
> - Different REST URI s can point to the exact same message in a service for example both POST /orders/ which creates an order where the server allocated the key; and PUT /orders/<order id> which creates an order were the client sets the order id can map to the same CreateOrder message which accepts an XML which may or may not have an order id.

- As REST is new to most SOA practitioners it is important to pay attention and avoid common REST mistakes like building a GETsful architecture (where only the GET api is used) not using hyper-media, using verbs as URIs (e.g. /createOrder/) etc.
- Lastly there's the point of whether or not you want to create a contract (e.g. using WADL) for the REST API. Personally, I think that if you have a proper REST API that utilize HATEOAS and properly implement the OPTIONS verb to allow checking for next steps it isn't really needed. Remember that the SOA API already has a more formal contract (event list or WSDLs) and that the REST API is supplementary.

SOA and REST can be made to work together and it is can be beneficial, especially if you plan to expose an API for consumption by UI application directly and not just for consumption of other applications. If you build your services properly, and employ REST practices namely use stateless communication, make results cacheable – you can, in fact, add REST as an additional API (or the only API for new services) and still get SOA's benefits.

So that's, REST - Let's see how does SOA match with another hot trend – the Cloud.

10.2 SOA and the cloud

Cloud computing is an important IT trend taking virtualization to the next level by using a large pool of virtualized hardware to provide utility computing capabilities. i.e. cloud computing provides an electricity-like mode where computational resources are available on-demand (usually with a pay-as-you-go model) and with the ability for elastically grow and shrink of resources use as needed.

We'd like to see how this, relatively new, playground affects SOA but let's first try to make sense of the different cloud related terms out there

10.2.1 The cloud terminology soup

Cloud computing sounds a lot like many other virtualization and hosting solutions we had around before. However, while cloud technologies share and use similar concepts there are several characteristics that differentiate clouds. The US national institutes of standards and technology published a formal definition of cloud computing (see further reading below) where they define five essential characteristics:

- **On demand self service**- the ability to provision additional capabilities (e.g. an additional VM, more storage etc.) by cloud users
- **Rapid elasticity** – As mentioned above, the ability to add/remove resources on demand
- **Measured service** – collecting, controlling, reporting and optimization of resources (bandwidth, CPU usage , etc.). The consumption of these resources is usually the basis for service charges

- **Resource pooling** – resources are shared by multiple consumers in a transparent manner i.e. users are not aware where exactly the resource are located or of other tenants using the resource.
- **Ubiquitous network access** – capabilities are accessed over the network and accessible via heterogeneous networks

Cloud computing can be delivered as a "public cloud" where anyone can register and use the resources e.g. Amazon AWS or Microsoft Azure. The pros of public cloud are

- low barrier to entry
- no up front investment
- a convenient pay-as-you go model
- virtually infinite scalability.

The cons include

- increased latency
- can be costly for the steady-state usage
- vendor lock-in (though that might be a temporary thing)

An alternative to public clouds, known as "private cloud", is to deploy a cloud on-premise for internal use by a single company. An example for that can be building a solution based on OpenStack or using VMWare vFabriq. Pros include performance and latency, familiarity of tools and technologies (for the cluster managers) and privacy/security. Cons include up-front investment, limited resources & scalability

Lastly there's the option of "hybrid cloud" i.e. using both a public and private cloud as a single solution. Hybrid clouds has the advantage of providing a good balance of flexibility and performance at the cost of increase. On the other hand hybrid clouds means more complexity and security challenges.

Cloud capabilities are, as mentioned above, delivered over the network. Cloud capabilities are delivered "as a service" and there are three main types of service delivery:

- Infrastructure as a Service – IaaS companies such as Amazon AWS, the cloud capabilities are basic building blocks like Virtual machines, storage, network bandwidth etc.
- Platform as a Service- PaaS cloud computing services, like Windows Azure, means that the cloud provider delivers infrastructure software components such as databases, queues, monitoring etc.
- SaaS – Software as a Service – these are usually smaller companies that deliver complete business capabilities e.g. Salesforce.com which delivers a CRM solution as a service

Ok, now that we've got the vocabulary sorted out, lets take a look at the architectural implications brought in by the Cloud.

10.2.2 Cloud and the fallacies of distributed computing

I mentioned Peter Deutsch's fallacies of distributed computing several times in this book and for a good reason. The fallacies are base architectural requirements that you have to account for when designing distributed systems. The cloud does not get a free ticket here:

Table 10.1 Fallacies of distributed computing and how are they relevant in cloud setups

Fallacy	What does it mean in the cloud
The network is reliable	No change this is still a problem. Especially in hybrid cloud solutions . If you have a real mission critical app you still need a disaster recovery plan e.g. backup in a secondary cloud provider.
Latency for zero	Latency has not decreased in the cloud. Though by deploying in datacenters near your end-users you can lower it. Cloud introduces another latency related problem
Bandwidth is infinite	In private clouds this hasn't changed from traditional systems. In public clouds it depends. For internal communications between deployed servers it has been transformed into a cost problem. For clients connecting to your cloud application it is same old problem
Topology doesn't change	If you assume this in a cloud solution you're in a real problem. The whole notion of elasticity means there's no way the topology stays the same.
There's one administrator	Like topology above, it is still a fallacy, just one that's hard to believe someone would make
Transport cost is zero	Still a problem, Cost ($) of moving data in an out of the cloud are more apparent than non-cloud environements as they come with a pricelist but the additional cost (performance, latency) on transforming data structures, encryption etc. can still be hidden.
The network is homogeneous	The network is not homogenous, however you don't care as much since you can define the types of machines you need and get virtualized copies that match your needs.

Table 10.1 shows that cloud computing doesn't solve distributed computing problem but at least it helps in making some of the fallacies more apparent so you're less likely to assume they're not there. However, the flip side, is that the cloud brings with it a couple new fallacies to watch out from:

- Nodes are fixed – This is a progression on the topology doesn't change fallacy and it means you can't assume too much on the node you are running on. Not its IP address, not that things you copied to it will be there on next boot – Don't assume anything. This means that any meaningful state should be persisted elsewhere on an attached or connected storage. On one project I worked on we had a service hosted in windows azure, we hosted this services in two district setups a staging and production (actually we had more for testing but that's less important). Windows azure has this feature where you can do a virtual IP switch to move the staging server(s) to production and we used it and it worked great – except the staging service was still pointing to the staging data store and using the staging certificate store – not so good. We solved this by orchestrating the whole switch from another service that also sent events to synchronize the whole move.
- Latency is constant – this is a progression on the latency is zero fallacy. The fact that latency is in fact not constant means that if you send messages in an asynchronous manner you can't assume they'd arrive in order. If you connect with user interfaces you need to understand the variance and plan for it so that users would get a decent experience. For instance, in the visual search service mentioned in chapter 9, we sometimes saw 5 to 15 seconds latency for establishing communications with the server to get a reasonable identification time we had to think about sending images/videos in the background before the user actually chose which image to identify.

Fine, but how does all this relates to SOA?

10.2.3 Cloud and SOA

SOA is **the** architectural style to enable transition to cloud computing especially for hybrid and public cloud scenarios. Table 10.2 below shows SOA effects and how they're a good fit for the Cloud

Table 10.2 SOA traits that are good fit for the cloud

SOA trait	How is good for the cloud
Partitioning of the enterprise/system into business components	A service is a good sized unit of to move to the cloud (as it is for moving to an external vendor). SOA component present a complete business function. Service boundaries already take into account the fallacies of distributed computing and already internalize handling of messages.
Standard based message and contract communications	Encapsulation of internal representations rather than relying on shared data means that services moved would be able to operate isolated from the rest of the world while communicating only via the

	messages defined in their contracts
Treating service boundary as trust boundary	When we want to move functionality to a public cloud it greatly helps if our software already assumes that anything foreign is hostile and should be authenticated, validated etc.
Keeping services autonomous	Autonomy better equips services to survive on their own, it also helps them to keep operating when other service go out

A lot of the patterns in this book are very relevant cloud deployments and even more so for the cloud transition phase, for example consider the following patterns:

- Service registry (chapter 7) so that other services know where to find services regardless if they're local or remote.
- Servicebus (chapter 7) also helps in providing the location transparency. Location transparency is very beneficial in cloud since new services might be spawn in new node with new ip address or consolidated to a single node based on load (elasticity)
- Identity provider pattern (chapter 4) is a crucial component when services are spread across the enterprise and a cloud and users expect a single-sign-on experience. This is even more important if you add REST (mentioned above) to the mix where you may need to interleave WS-trust and oauth services
- Request/Reaction and Inversion of communications (chapter 5) – asynchronous communication is more resilient than plain RPC and that's a bug plus esp. in hybrid cloud setups
- Monitoring and watchdog (chapter 3) – always relevant but even more important when you don't control the hardware
- Service instance (chapter 3) another pattern that can help with elasticity and scale out
- Virtual Endpoint (chapter 3) When running in the cloud, the endpoint in which services are delivered will most likely be a virtual endpoint whether we like it or not.

In summary, SOA principles and patterns are a very good match for the cloud. The division of business capabilities into autonomous components fit well both gradual transitioning to public clouds as well hybrid cloud setups.

10.3 SOA and big data

I remember seeing this interesting video a few years ago called "Shift Happens" which includes all sorts of interesting trivia on the rate the world is changing in the digital age. The 2010 version of this video includes an estimation that 4 exabytes (10^19) of unique information was generated that year (which is more than the previous 5000 years

combined). Excluding the Googles and Facebooks of the world most of us don't have to deal with these amounts of data but still there's no denying that the amount of data enterprises have to process and amass every year continuously grows. For instance a TDWI research from sept. 2011 says that a third of the organizations surveyed had more than 10 terabytes of data and that number of larger sets (100s of terabytes) will triple in 2012.

Most research organizations (like TDWI mentioned above or Forrester research) agree that big data evolves around different types of "V"s – like Velocity, Volume, Variety and Variability. Personally I think the major drivers are just the first two Vs – the Velocity in which you have to ingest data along with the latency you can afford to take until it is usable and the total volume of data you want/have to store and actually do something with. So for example if you have a high-peak load of messages for a couple of hours a day and then you only need to see that data a day later – that's not a big data problem. Same goes to terabytes of archive data you don't analyze and just store there for say regulatory reason.

Big data, has a lot of implications starting from changing the way we think about data , the emergence of new professions like data scientists and it also has technical implications which is what we'll take a quick glimpse at next.

10.3.1 The big data technology mix

I read a blog post by Gil Press that stated that the first big data problem was in the 1880s (yes you read that right). In the late 1800s the processing of the US census was beginning to take so long that it was getting close to 10 years. Crossing this mark is meaningful as the census runs every 10 years and as birth rates are getting higher the outlook wasn't very good. In 1886 Herman Hollerith started a business (that year later was merged with other companies to form IBM) to sell a tabulating machine that holds census data on punch cards. Indeed the 1890 census took less than 2 years to complete and handled both larger population (62 million people) and more data points than the 1880 census.

Today we find ourselves in similar position when we're looking at big data problems and try to solve them with the traditional tools we have at hand like our trusty RDBMS or OLAP cubes. It isn't that any of them are going away, but we need additional tools, our own Hollerith machines to cope with the scale. The good news is that a lot of these new tools are emerging. The bad news is that **a lot** of these new tools are emerging. Figure 10.3 shows some of the main categories of solutions for big data storage, that emerges in the marker and a few examples of tools in each category. For instance there's the Relational category that is divided between NewSQL solutions (sharding solutions over regular RDBMSS) and Massively Parallel solutions. The massively parallel solutions are then divided into column oriented solutions and row oriented ones. On the other side of the scope we have key-value stores which are divided between in memory and column oriented etc. The diagram is not exhaustive but it does demonstrate the wide range of options and sub-options available, and in a sense, that there's no single good solution - otherwise there'd be less options and

everyone would standardize around the best solutions (as it happened with RDBMSs 30 years ago)

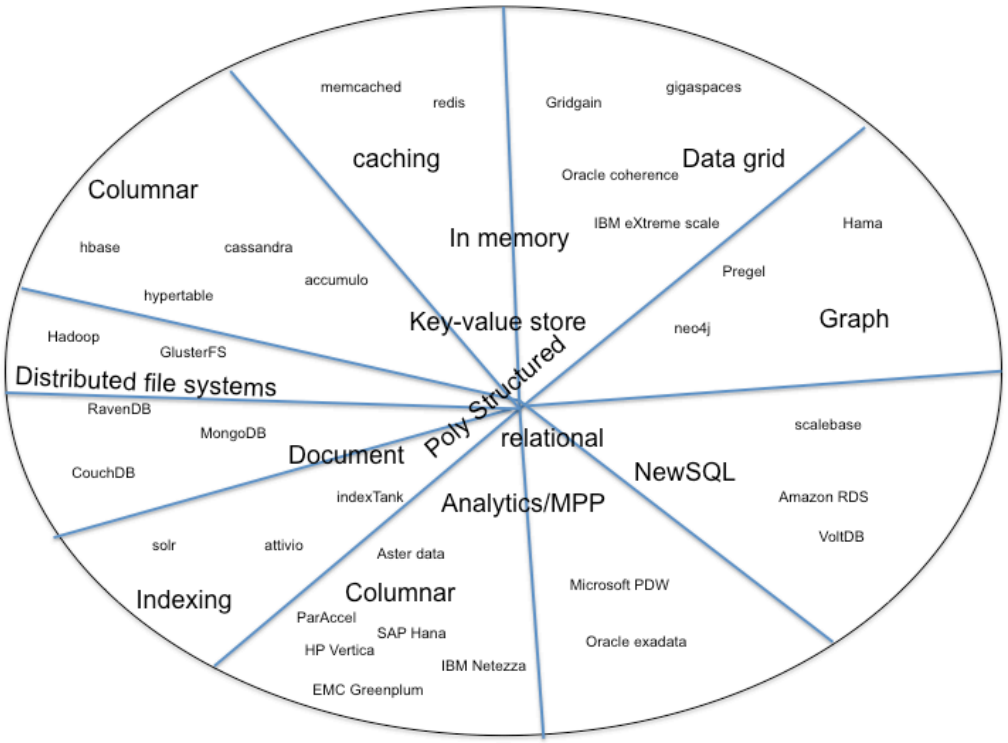


Figure 10.3 The big data storage space. There are several classes of solutions some are based on the relational paradigm and some remove some or most of the database capabilities and to get massive scale at cheap prices

With an almost endless poll of option to choose from we need some selection criteria to choose the best solution for a given project. Below are some of the criteria I find useful.

CRITERIA TO CHOOSE THE BEST SOLUTION

- Type of organization – Enterprises will likely be more drawn to the more established vendors (for support, regulatory compliance etc.). Startups will most likely gravitate toward the cheap, open source options.
- Data access patterns – Mostly reads vs. mostly writes, access based on primary key or a lot of ad-hoc queries. E.g. if you need to traverse relations back and forth (like walking a social graph) – graph databases can be a good option.

- Type of data stored – structured – good fit for relational models, semi structured (XML/JSON) which is a good fit for document and column stores, unstructured good for file based options like Hadoop
- Data schema change frequency – mostly fixed vs. constantly changing
- Required latency – the faster you need the data the more you'd want/need a in-memory solution

APACHE HADOOP There are a lot of interesting technologies in the big-data space, one that stands out, though is apache Hadoop. Apache Hadoop is an open source implementation of Google's file system and map/reduce paradigm. Hadoop is interesting, not because it is necessarily the best solution for big-data, but rather since it has gained a massive backing from many of the major IT vendors Oracle,IBM, EMC, Microsoft, Amazon all of which offer an Hadoop distribution or service. I've included a few sources about apache Hadoop in further reading section of this chapter.

At this point you might be thinking that big data sure sounds interesting but where's the place for SOA in all this. How can we fit SOA into all of this?

10.3.2 Any SOA there?

Is there any SOA in big data? Well, the right question is actually a little different, it is more "how can SOA work with big data?". If we accept the premise made at the beginning of this section that more and more enterprises find they need to handle big data, then SOA should be able to work with that or be replaced with a more appropriate architecture.

One way for adoption is that services will use big data related technologies within the services. A service that needs to handle semi-structured data can use a document database store and another that needs to handle event data in near real-time will use a data grid or an event stream processing solution. Like with cloud technology, the advantage of SOA is the separation and isolation of the various services from one another. The isolation allows gradual adoption in the enterprise where only services that need these technologies adopt them where other services can stay with their current technologies.

One related pattern here is the gridable service (see chapter 3) that describes taking a computational intensive task and dividing it between multiple services – something you can achieve with both data grid solutions as well as big data stores that support map/reduce.

When it comes to analysis of big data we should distinguish between the places where the analysis can be made within the boundaries of the service and when the analysis requires data from multiple services.

For the second type of big data analysis, where a cross-service view is needed, the ideas described in the aggregated reporting pattern (see chapter 7) still apply. We can get the data from all the services in a way that does not violate SOA principles as long as we make the data immutable and we know where the ownership lies. The processes that perform the

actual analysis can sometimes be considered services themselves e.g. a recommendation service for ecommerce solutions.

When the analysis can be handled within the boundaries of a specific service implementation is a matter of utilizing big data related technologies as part of the service. For instance in a system I recently worked on we had to do categorization of mutli-channel interactions (voice, email, chat etc.). The categorization service had a subscription for incoming interactions, which arrived both in batches and in real-time. The same business logic that categorized data in real-time was also used in the batch. the real-time categorization had a web service and messaging endpoints and the batch processing used map-reduce on top of Hadoop – two parts of the same business service, using the same business logic to do their work. Figure 10.4 below provides an illustration of this service.

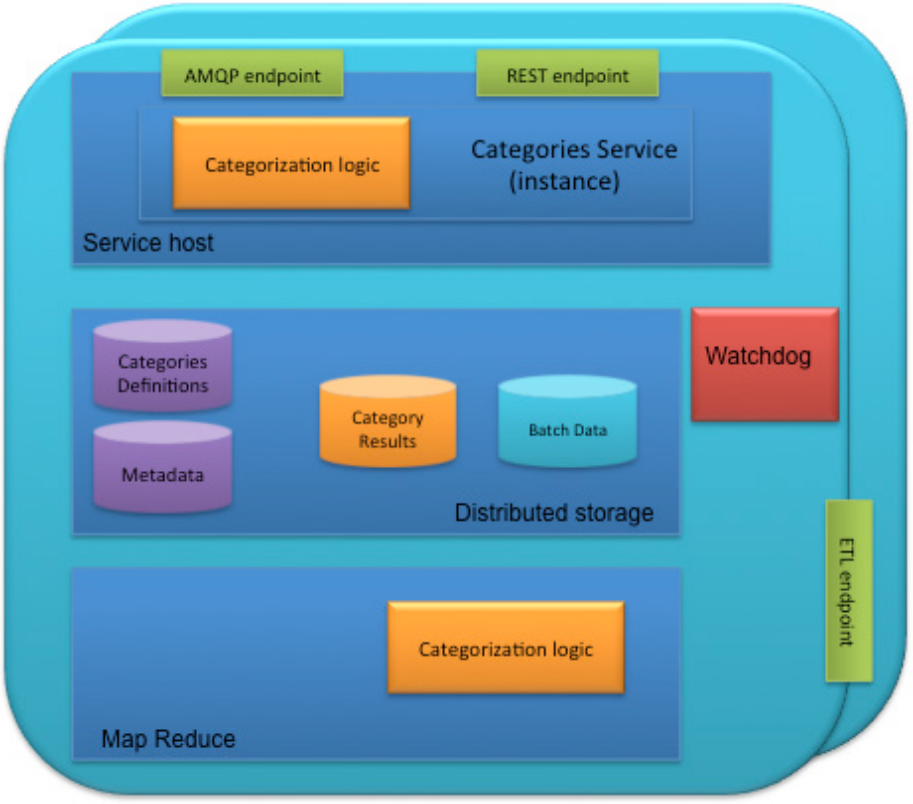


Figure 10.4 an illustration of a categorization service that incorporate big-data map/reduce handling with on-line handling. The service has 3 endpoints an ETL one that enables ingesting large batches of updates, a REST end point that accepts small batches and on-line requests and an AMQP endpoints for low-latency requests. The same categorization logic is reused between the map/reduce batch processes and the on-line/real-time processes.

In addition to the specifics around big data, we can see the application of some of the patterns described in this book within the illustration. For instance the service host (chapter 2) that hosts the service with its two endpoints – each of which is an Edge component (chapter 2). Note that one of the endpoints is a RESTful one as discussed earlier in this chapter. Additionally we see the Watchdog (chapter 3) and that the service is deployed multiple times (service instance from chapter 3)

In summary, we've seen that Services can be used with big data. Big data emphasizes the need to make service coarse grained (see also discussion on nano-services anti-pattern chapter 8) – also what we learned on building services is still applicable – nevertheless, Big data is changing the way enterprises handle and think about data. For SOA to stay relevant as an architectural style it should – and can, adapt and also utilize the new technologies that solve big-data problems.

10.4 Summary

There are of course other architectural styles and technologies that are or can be related to SOA, for instance we discussed Event Driven Architecture (EDA) and SOA as part of the inversion of communications pattern in chapter 5. Another relevant style is domain driven design, which isn't as popular as the three trends discussed here but can complement SOA as a way to design individual services. The styles we did cover in this chapter include:

- REST – an alternative architectural style which can be merged with SOA. If you build RESTful SOA you can benefit from both and use either (or both) SOA style or REST style APIs for your services
- Cloud – a complementary IT trend that shares it principles with SOA and for which SOA is a very good fit.
- Big data – a reality that is becoming common in a lot of enterprises and to which SOA has to adapt.

Congratulations on finishing the last chapter. You should now be able to understand the main challenges and common pitfalls of building distributed systems in general and service oriented ones in particular. You should also have an arsenal of architectural concepts that let you cope with these challenges and build solid systems. The book took quite a few years to complete – mostly because I was too busy building systems to complete it. I'd like to think that the book reflects this and that it is a practitioner guide even though it deals with architecture, which in essence is more abstract than code.

The focus of this book, as mentioned above, is SOA as a way to solve distributed systems challenges, so naturally, the coverage of other architectural styles only scratched the surface. You can take a look at the further reading section below for resources expanding on the topics mentioned here.

10.5 Further reading

The further reading section of this chapter is probably the largest in this book. The reason for that is all the topics discussed here are as big and diverse as SOA. They warrant and have their own books, articles and we've just glimpsed into them here.

Table 10.3 resources for further reading on topics covered in this chapter.

Topic	Resource name/link	Why
REST	Roy Thomas Fielding 2000, "Architectural Styles and the Design of Network-based Software Architectures" , http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm	Roy dissertation is the first to define the REST architectural style
REST	Jim Webber et. Al. "How to GET a cup of coffer" http://www.infoq.com/articles/webber-rest-workflow	A good explanation of REST principles including HATEOAS
REST	RESTful web services Leonard Richardson, Sam Ruby http://www.amazon.com/Restful-Web-Services-Leonard-Richardson/dp/0596529260	Probably the best book on REST
Cloud	The Cloud at Your Service. The when, how, and why of enterprise cloud computing. Jothy Rosenberg and Arthur Mateos http://www.manning.com/rosenberg/	A good all around introduction to cloud
Cloud	OpenStack – open source cloud platform http://openstack.org/	An open source cloud implementation
Cloud	ReaderWriterWeb -http://www.readwriteweb.com/cloud/	a ReadWriteWeb channel dedicated to cloud
Big data	Hadoop in practive Alex Holmes	An updated view of Hadoop and related technologies
Big data	HBase the definitive guide, lars George http://ofps.oreilly.com/titles/9781449396107/index.html	Best book on HBase
Big data	*TDWI research* "Big data analytics Q42011" http://tdwi.org/research/2011/09/best-practices-report-q4-big-data-analytics.aspx	Overview of the big data landscape

Big data	http://nosql-database.org/	a site with links to a lot of no sql databases as well as articles
Big data	DBMS2 – Curt Monash's blog http://www.dbms2.com/	A good blog database and related technologies
Big data	*myNOSql – Alex Popescu's blog* http://nosql.mypopescu.com/	A blog on NoSQL news
Big data	Marco on CEP – Marco Seiriö blog on complex event processing http://rulecore.com/CEPblog/	A blog on complex event processing technolgies

10.6 Bibliography

NIST definition of cloud computing http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

Fallacies of Distributed computing http://blogs.oracle.com/jag/resource/Fallacies.html

Shift happens 3.0 http://www.youtube.com/watch?v=cL9Wu2kWwSY

TDWI research "Big data analytics Q42011" *http*://tdwi.org/research/2011/09/best-practices-report-q4-big-data-analytics.aspx

The birth of big data Gil Press http://infostory.wordpress.com/2011/06/15/the-birth-of-big-data/