1 Introduction 4
2 Mandatory Standards 5
2.1 WSDL 5
2.1.1 Service Definition Layer 5
2.1.2 Binding Layer 6
2.2 SOAP 7
2.3 UDDI 8
2.3.1 Different types of UDDI registries 8
2.3.2 Interacting with UDI 8
2.3.3 Publishing information 9
2.3.4 Finding information 9
2.3.5 Using the obtained information 9
3 Choosing the right web service style 11
3.1 Introduction 11
3.2 Document style 11
3.3 RPC style 11
3.4 Best practices 11
3.4.1 Design with validation in mind 12
3.4.2 Design with interface compatibility in mind 12
3.4.3 Take into account the invocation model 12
3.4.4 Understand the differences in implementing the service 12
3.4.5 Take into account the statefullness of services 13
3.4.6 Design with Interoperability in mind 13
4 Choosing the right web service usage 14
4.1 Introduction 14
4.2 Best practices 14
4.2.1 Design with interoperability in mind 14
4.2.2 Performance & Scalability 14
4.2.3 Try to look ahead 14
5 Choosing the right binding model 15
5.1 RPC VS Document 15
5.2 Encoded VS Literal 15
5.3 Choosing the binding model 15
5.3.1 Binding model effects the service Interface 15
5.3.2 Binding model can effect the way validation occurs 16
5.3.3 Binding model can have an impact on performance 16
5.3.4 Conclusion 16
6 Service Transport Layer 17
6.1 HTTP 17
6.1.1 Advantages 17
6.1.2 Disadvantages 17
6.2 JMS 18
6.2.1 Advantages 18
6.2.2 Disadvantages 18
6.3 SMTP 18

6.3.1 Advantages 18
6.3.2 Disadvantages 18
7 Service invocation model 19
7.1 Synchronous invocation 19
7.2 Asynchronous invocation 19
8 Message Exchange Patterns 21
8.1 SOAP Message Exchange Patterns 21
8.1.1 SOAP Request Response 21
8.1.2 SOAP Response 21
8.1.3 SOAP Multicast 21
8.2 WSDL Message Exchange Patterns 21
8.2.1 input-output 22
8.2.2 output-input 22
8.2.3 input only 22
8.2.4 output only 22
8.3 WS-MessageDelivery Specification 23
9 Service Interface Specifications 24
9.1 Loosely typed VS strong typed 24
9.1.1 Loosely typed service 24
9.1.2 Strong typed service 24
9.2 Best Practices 25
9.2.1 Interfaces should be strong typed 25
9.2.2 Interfaces should have meaningful parameters 25
9.2.3 Interfaces should be document centric 26
9.2.4 Interfaces should be compliant with WS-I Basic Profile 26
9.2.5 Interfaces should be rich enough to shield the user away from implementation changes 26
9.3 Conclusion 26
9.4 Service Interface Granularity 26
9.4.1 Conclusion 28
9.5 Avoiding the use case mismatch 28
10 Service versioning 30
10.1 Using XML Schema for versioning 30
10.2 Using UDDI 31
10.3 Running multiple versions of the same service 32
10.4 Service Release Management 32
10.5 Leverage vendor offerings 33
11 Service development 34
11.1 Contract First approach 34
11.2 WSDL Predefined extensions. 34
12 Schema handling 35
13 Securing Web Services 36
13.1 Transport Layer Security 36
13.1.1 SMTP 36
13.1.2 HTTP 36

13.1.3 HTTPS 36
13.2 Service Description Layer Security 37
13.2.1 XML Digital Signatures 37
13.2.2 XML Encryption 37
13.2.3 Security Assertion Markup Language (SAML) 37
13.3 Web Service Security Standards 37
13.3.1 WS-SecureConversation 37
13.3.2 WS-Federation 37
13.3.3 WS-Authorization 38
13.3.4 WS-Policy 38
13.3.5 WS-Trust 38
13.3.6 WS-Privacy 38
14 Interoperability 39

1 Introduction

This section of the document will outline some design principles and best practices when it comes to service design. We'll cover several aspect that any service oriented architecture should take into account. This includes

- Mandatory standards
- Service interface design
- Service versioning
- Data Modeling
- Interoperability

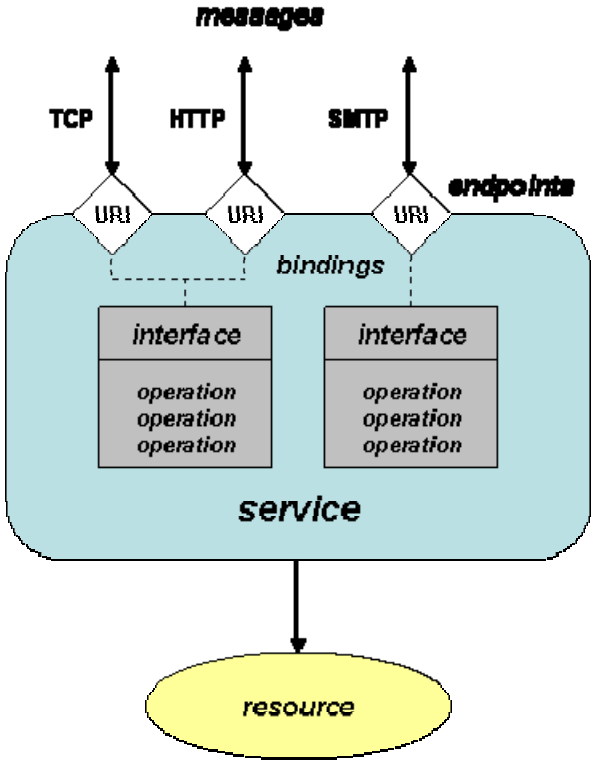
2 Mandatory Standards

2.1 WSDL

We've already seen that a WSDL (Web Service Description Language) document describes a service. We can look at it as an interface description language for services.

A WSDL document contains 2 distinct parts

- Service Definition Layer (Interface layer)
- Binding Layer



2.1.1 Service Definition Layer

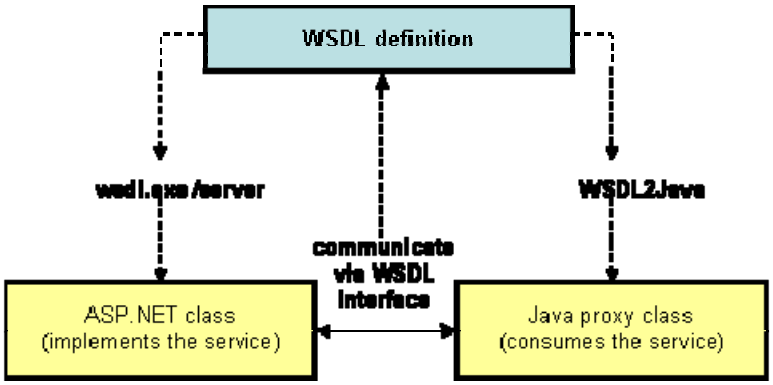
The **Service Definition Layer** in WSDL basically defines

- The public interface (methods / operations) that a service exposes

- The arguments and return types (messages) associated with those methods
- The data model used by the service

WSDL basically defines the contract between the consumer and the producer. The benefit of using WSDL (besides the fact that it's an ideal language to describe services), is that it provides a formal description based on standards (XML, XML Schema …) resulting in excellent tool & vendor support.

Both Microsoft and Java provided utilities that are able to generate client side code (stubs) based on a WSDL file. This way, the developer targeting web services can avoid much of the plumbing code involved in working with web services (sending / receiving SOAP messages, connecting to endpoints …)



2.1.2 Binding Layer

The **Binding Layer** in WSDL describes the binding between the service, and the underlying messaging protocol (primarily SOAP), as well as the data formats used.

The SOAP binding in WSDL defines

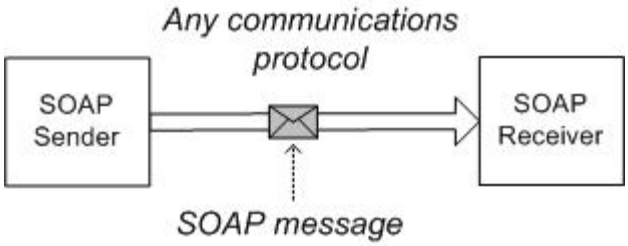
- 2 binding styles (message encoding , what does the soap body look like)
  - RPC
  - Document.
- 2 types of usages (encodings)
  - Encoded
  - Literal
- 2 types of protocols
  - HTTP
  - SMTP

2.2 SOAP

SOAP (Simple Object Access Protocol) is basically a framework for exchanging XML based messages between various business partners.

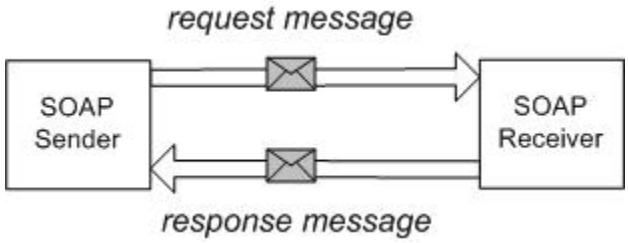
SOAP defines a processing model that outlines how a message gets processed between the consumer and the producer.

In its simplest form, a SOAP interaction between the sender and receiver will look like this.



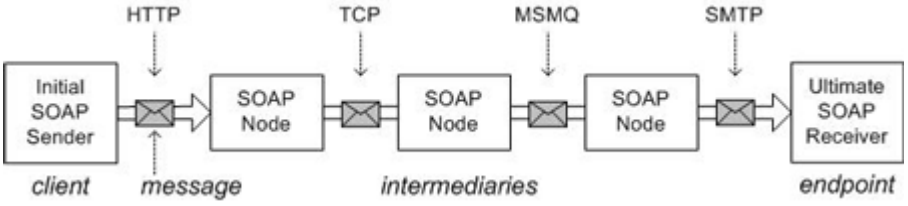
Simple one way messaging

However, other models are possible, for example



Request/Response

Multiple intermediate nodes can be introduced in the architecture, allowing for the following architecture



SOAP message traversing multiple intermediate nodes

The intermediate nodes depicted above acts as both a SOAP receiver and SOAP sender at the same time. A SOAP node can have one or more roles. The role

determines how the message will be further processed down the chain. SOAP 1.1 defines a single role called '**next**' (http://schemas.xmlsoap.org/soap/actor/next). When a message arrives at a SOAP node, it must therefore process the mandatory headers in the message. (As it assumes the '**next role'**).

2.3 UDDI

In any service oriented architecture, we need some kind of registry that is used to acts as a container for the various services. UDDI (Universal Description, Recovery and Integration) provides this registry mechanism so that clients can easily find the services they're interested in.
It basically provides a registry API (based on WSDL / SOAP) for registering and discovering web services.

2.3.1 Different types of UDDI registries

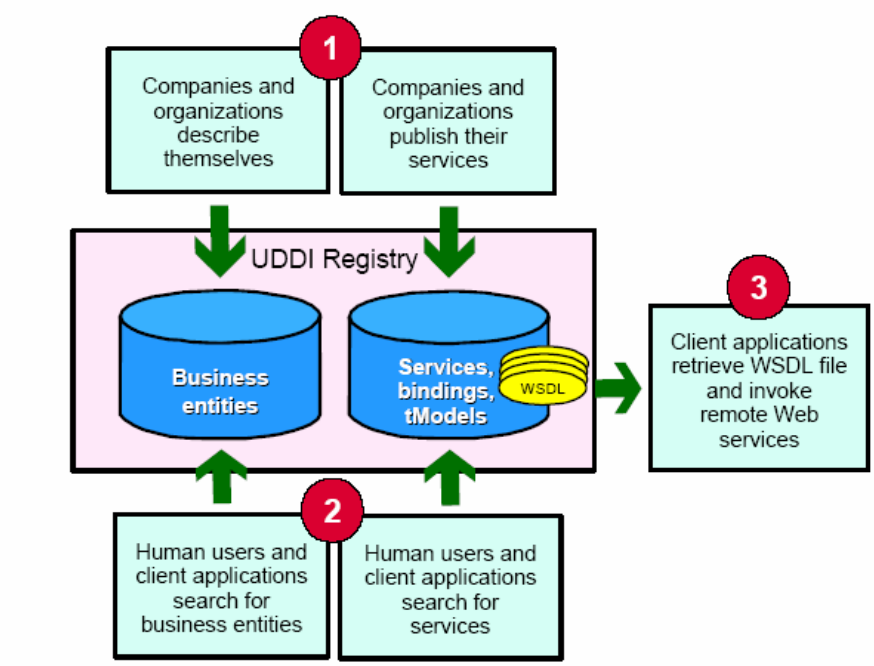
- Public UDDI Registries
- Private UDDI Registries

[discuss pros/cons & usages]

2.3.2 Interacting with UDI

The primary interactions that a private person or an application client will have with UDDI are:

1. Publishing information
2. Finding information
3. Using the obtained information



Interactions with UDDI

2.3.3 Publishing information

The following entities can be published into a UDDI registry

- Business entity information
- Business service information
- Binding templates
- tModels
- Taxonomy information
- Publisher assertions (business relationships)

2.3.4 Finding information

UDDI offers various interaction points for either humans / application clients.

- Most UDDI registries offer some kind of **HTML based user interface** so that **human parties** can interact with the registry.
- UDDI also exposes an **API** so that **application clients** can programmatically publish/find and use information coming from the UDDI registry.

2.3.5 Using the obtained information

In order to use the information obtained from the registry, the application client will use the interface described in the WSDL document to execute a method / send an

XML message to the web service endpoint witch is also defined in the WSDL document.

3 Choosing the right web service style

3.1 Introduction

When designing web services, one of the first choices the implementer will have to make is deciding whether to use a RPC or Document centric design.

The following table outlines the primary differences between the 2 designs:

RPC	Document
Execute function calls	Exchange documents
Payload is defined by an external schema (external encoding rules)	Payload (XML) is defined by an XML Schema
Fine grained operations	Coarse grained operations
Promotes synchronous programming model	Promotes Asynchronous programming model
Modeled as remote procedure calls	Modeled using XML Document Exchange

The basic difference between the 2 models is the way requests are issued to a service provider.

3.2 Document style

Document style web services are all about exchanging messages.
send(document)

```
<out:getNoteResponse xmlns:out="urn:outline.demo">
   <out:note key="000000000B" >
      <out:content>test</out:content>
   </out:note>
</out:getNoteResponse>
```

3.3 RPC style

RPC style webservices are all about invoking a method on a remove entity.
someOperation(parameters[])

```
<tns:matchNoteAndNote xmlns:tns="urn:outline.demo">
   <in0 xsi:type="xsd:string">0000000000</in0>
   <in1 xsi:type="xsd:string">000000000B</in1>
</tns:matchNoteAndNote>
```

3.4 Best practices

Now that we have discussed the differences between Document style & RPC style web service, we'll now look at some arguments that will help you decide between the 2 models.

3.4.1 Design with validation in mind

- Using Document style web services, you can describe and validate a high-level business document using XML.
- RPC style web services only have the method name and parameters in their payload, so no high level business rules can be enforced

3.4.2 Design with interface compatibility in mind

- RPC calls are considered static (changes in the method signature breaks the contract with the client)
- XML schemas are more flexible (they can change more often by adding elements without breaking the contract with it clients

3.4.3 Take into account the invocation model

Document style web services are better suited when it comes to asynchronous processing. The ability to do asynchronous processing within the Service Oriented Architecture helps us to create added value when it comes to:

- Reliability
- Scalability
- Performance
- Fault tolerance

Reliable messaging usually occurs through some kind of middleware (persistence queues, store and forward principle).The transport & routing can be delegated to the middleware asynchronously, leaving the service capable of handling subsequent requests.

3.4.4 Understand the differences in implementing the service

The effort to design a Document Style web services is more elaborate then designing an RPC style web service.

In order to implement a Document style web service, we need to

- Design an XML schema
- Support an existing XML schema
- Interpret the response message , and filter out the data required

In order to implement an RPC style web service, we need to

- Execute a method
- Interpret the result

Although it's easier from a developers point of view to create RPC style web services, most of the complexities involved in creating Document style web services can be

delegated away from the developer, and onto the many SOAP toolkits that are available.

3.4.5 Take into account the statefullness of services

If maintaining state is a requirement, document style web service are far better suited then RPC style. It's considered very difficult to implement this using RPC. XML Documents on the other had can carry the state around far more easily.
External services (used by external entities) should always be document style

3.4.6 Design with Interoperability in mind

In order to guarantee interoperability between the various components in the architecture, web service design should take into account the WSI Basic Profile (more on that topic in a subsequent section). The WSI Basic Profile allows for both RPC and Document style web service to be used within the architecture.

4 Choosing the right web service usage

4.1 Introduction

Encoded VS Literal is all about how the consumer & producer interpret a given set of messages. It determines how the data types are represented in the XML message payload.

- In the case of **Encoded usage**, a contract is defined up front, using an encoding scheme. (encoding rules can be found in the SOAP specification)
- In the case of **Literal usage**, an XML Schema is provided to define the data types.

SOAP was originally targeted as a solution for integrating distributed object technologies (such as DCOM, RMI & Corba) with internet technologies such as HTTP/XML.
As SOAP handles the lower level plumbing of serializing these objects into XML format, it needs to provide some help in interpreting those XML messages. The obvious chose today would be offcourse to use XML Schema to handle this, but at the time of the original SOAP specification, XML Schema wasn't an option at the time, as it was far from completed.

SOAP Encoding is something that stems from the past. We would advice against the use of SOAP Encoding for a variety of reasons.

4.2 Best practices

4.2.1 Design with interoperability in mind

In order to achieve interoperability, and compliance with the WSI Basic Profile specification, one should abandon the use of SOAP encoding. The WSI Basic Profile formally forbids the use of SOAP Encoding.

4.2.2 Performance & Scalability

Several benchmarks have concluded that there is a performance & scalability penalty involved when working with SOAP RPC encoding, especially when the payload of the message increases.

4.2.3 Try to look ahead

- SOAP 1.2 has made SOAP encoding optional, meaning that SOAP Toolkit vendors can get certified without providing support for SOAP encoding.
- WSDL 1.2 has chosen to drop support for SOAP encoding

5 Choosing the right binding model

5.1 RPC VS Document

RPC VS Document talks about how a WSDL binding should be translated to a SOAP message. It describes how the actual payload of the message looks like over the wire.

- In the case of **RPC style**, the message contains the information with regards to the operation that is to be executed (method name , arguments, …)
- In the case of **Document style**, all communication is done using XML messages

5.2 Encoded VS Literal

Encoded VS Literal is all about how the consumer & producer interpret any given set of messages. It determines how the data types are represented in XML.

- In the case of **Encoded usage**, a contract is defined up front, using an encoding scheme. (encoding rules can be found in the SOAP specification)
- In the case of **Literal usage**, an XML Schema is provided to define the data types.

This gives us the following **WSDL binding models**

- RPC/encoded
- RPC/literal (*)
- Document/encoded
- Document/literal (*)

(*) Note that the WSI-Basic Profile only allows for Document/Literal and RPC/Literal usages.

WSDL includes a **binding element** that supports 2 attributes in order to represent the 4 options above

- Style attribute of the binding element
- Use attribute of the binding element

5.3 Choosing the binding model

5.3.1 Binding model effects the service Interface

- We've seen that the service interface is the only true contract between a service consumer & producer. Service interfaces tend to change over time, so we need a flexible way of dealing with these changes.

- RPC based service interfaces should be fairly static in nature, as any change in the service interface will effectively break the contract between the consumer and producer.
- When using a document/literal style, the impact of changing the service interface can be minimized. (Late binding pattern)

5.3.2 Binding model can effect the way validation occurs

- When using a document centric approach, XML validation can be used to enforce certain high level business rules.
- These high level business rules are defined in XML Schema. The generated XML can be validated before invoking an operation on the service producer.
- If the service depends on complex xml structures, a document centric approach is the better choice.

5.3.3 Binding model can have an impact on performance

- When working with XML, we know that at some point we'll need to perform some kind of XML parsing. This is typically considered to be an expensive operation, especially when the DOM tree reaches a certain limit.
- Document style web services can take advantage of other XML parsing techniques such as SAX to have a smaller memory footprint, and better overall performance.

5.3.4 Conclusion

- Due to the fact that the WS-I Basic Profile only allows Document/Literal and RPC/Literal usages, web services should be developed using one of those style/usage combination.
- SOAP encodings are not allowed in the Basic Profile, so in order to insure interoperability, these usages should be avoided.

6 Service Transport Layer

The service transport layer encompasses the protocol that is used to get the messages from the consumer to the producer and vice versa.
A web service, once exposed on the message bus, can have many bindings to various protocols. Deciding on the appropriate binding for a particular client is vital for the performance of the overall architecture.

Typically, an organization will have many protocols already in place to perform messaging. Usually some kind of message oriented middleware will be in place. Other available protocols might include HTTP, HTTPS, JMS, MQ, and SMTP …
The WSDL format allows for a particular service to be bound to several of these protocols. This means that a single service could be accessible via HTTP and JMS, depending on the implementation of the service.

Protocol	Invocation model	Data format	Description
HTTP	Synchronous / Asynchronous	XML SOAP	Remote, unsecured, platform independent, service access within or external to org.
HTTPS	Synchronous / Asynchronous	XML SOAP	Remote, secure, platform independent, service access within or external to org.
JMS	Asynchronous	XML SOAP	Peer to peer or published messages in communicating Java environment
SMTP	Asynchronous	Context Based XML SOAP	Remote delivery of messages within or external to org

6.1 HTTP

6.1.1 Advantages

- Widely adopted protocol
- Open protocol
- Enabled to pass through firewalls

6.1.2 Disadvantages

- Stateless protocol
- Not reliable

6.2 JMS

6.2.1 Advantages

- More reliable means of transport (as opposed to HTTP)
- Automatic support for asynchronous messaging
- Enterprise proven messaging technology

6.2.2 Disadvantages

- JMS is java based and will only run out of the box on other platforms

6.3 SMTP

6.3.1 Advantages

6.3.2 Disadvantages

[discuss smtp]

7 Service invocation model

In a Service Oriented Architecture, one can make use of 2 invocation models

- Synchronous
- Asynchronous

7.1 Synchronous invocation

When immediate feedback is require upon a service method call, one should use synchronous messaging.

This puts some stress on the service producer, as he will be required to

- Return some kind of response within a predefined time span.
- Have some level of performance allowing it to handle a certain load
- Be up & running during the period where service invocations can occur

In terms of the service consumer this means that

- It will have to wait until a response is received from the service producer
- No further processing on the client side can occur before a response has been returned from the producer.
- When using the asynchronous model, the service doesn't need to send a response to the client immediately. The client can invoke a service method, and will continue with its processing. This model is typically implemented by using some kind of middleware solution (queues / topics). Messages can be queued so that they can be picked up by the service producer whenever the service is available.

7.2 Asynchronous invocation

Using the asynchronous model implies that

- The service doesn't have to be up & running during a service invocation. Messages can be queued when the service is down , or is experiencing heavy load, and later processed when the service is up & running
- The client won't receive immediate feedback, but can continue with other processing.

Use asynchronous messaging when

- A large number of client requests can potentially invoke a particular service, but the service implementation can only handle a limited number of simultaneous requests.
- No immediate feedback is required from the service.

- The client needs to continue with its processing after the service invocation.

In order to use asynchronous messaging, the overall architecture will need to have some mechanism for reliable messaging. This is typically implemented using middleware solutions such as queues and topics. In that respect, the Enterprise Service Bus can acts as a layer of abstraction for hiding away the complexities of interacting with those queue managers.

Asynchronous messaging is often referred to as sending 'fire & forget' messages. The consumer isn't necessarily interested in a response from the server.

In the case where a response is required, the response message will also be delivered to the consumer in an asynchronous way (delivering a message on a certain queue that can be picked up by the original sender of the request, sending an email message …)

8 Message Exchange Patterns

A Message Exchange Pattern describes the type of message flows (and associated processing) between 2 parties. Message Exchange Patterns are available both on a SOAP level, as well as on the WSL level.

8.1 SOAP Message Exchange Patterns

8.1.1 SOAP Request Response

This pattern represents the classical request response model, where party A sends a message to party B, and expects a response back. Both messages are formatted using SOAP.

8.1.2 SOAP Response

The SOAP Response MEP allows us to work with GET operations on the HTTP protocol. This MEP basically involves one single SOAP messages in response to a non-SOAP message. This pattern is typically implemented by embedding some kind of URL in a previous SOAP interaction. The URL can perform an HTTP GET operation , and doesn't involve sending a SOAP message.
The client will issue a GET request and sets the Accept header to **application/soap+xml** in to request a SOAP answer.

Performing GET operations should be only be done with requests that are safe & idempotent

- Safe: All we are doing is retrieve information.
- Idempotent: different requests should always return the same result

[insert scenario here … URL embedded in a previous SOAP response, client uses the URL to perform an HTTP GET operation and expects a SOAP message as a response]

8.1.3 SOAP Multicast

This pattern describes simultaneously sending messages to various parties.

8.2 WSDL Message Exchange Patterns

On the WSDL level , we also have a notion of Message Exchange Patterns , although in the WSDL realm we prefer to refer to them as just 'patterns' (in order to differentiate with the SOAP MEPs).

WSDL Patterns define the following aspects of messaging:

- Sequence
- Direction

- Cardinality

The WSDL Patterns don't define

- The type of messages being sent
- Invocation model (synchronous/asynchronous)
- Timing between messages

The WSDL patterns describe message interactions at the abstract interface level. These don't necessarily have to map to the patterns that are defined by the underlying protocol (HTTP, JMS). For example, the request-response MEP is something that's offered out of the box by the HTTP implementation, but a typical one way protocol like JMS could also implement the request-response.

WSDL 1.1 initially supported the following patterns

- input-output,
- output-input,
- input only
- output only

Those roughly correspond to the following **message interaction patterns**

- Request-response
  The endpoint receives a message , and sends a correlated message
- Solicit response (aka publish-subscribe)
  The endpoint sends a message , and receives a correlated message
- One-way (input only)
  The endpoint receives a message
- Notification (aka event)
  The endpoint sends a message

8.2.1 input-output

This is the Request Response pattern

8.2.2 output-input

8.2.3 input only

8.2.4 output only

[Discuss these MEPS further

WSDL only defines bindings for request-response and one-way. The Solicit Response and Notification operations have no popular bindings.

[Elaborate]

WSDL 1.2 extended those messaging patterns with the following set:

- input only
- input-output,
- request-response,
- input multi output
- output only (notification)
- output-input (solicit response)
- output-multi-input
- multicast-solicit-response

8.3 WS-MessageDelivery Specification

[discuss the WS-MessageDelivery Specification briefly]

9 Service Interface Specifications

9.1 Loosely typed VS strong typed

A loosely typed service is a service that doesn't impose a lot of restrictions on the message format that needs to be exchanged between a consumer and a producer. The actual semantics of the messages are handled on the service implementation level.

9.1.1 Loosely typed service

An example of a loosely typed service method signature could be:

```
public String execute(String request)
```

This signature defines a single method that accepts a String as an input parameter, and a String as a return value. From an outsider's point of view, there is no way of knowing the exact semantics of this operation. The String can contain pretty much anything.

Although this might seem like a flexible approach (no schema constraints / changes to deal with) it does impose extra work for both the consumer and the producer of the service.
The WSDL definition is not sufficient in order to define a contract between the consumer and the producer. Although it does provide some sort of contract between the consumer and producer on the interface level, the consumer still needs to know the exact semantics of the input and output parameters for this service. This behavior or logic needs to be present in the client side code.
Both consumer and producer need to know how to interpret these Strings, and both implementations need to be aligned.
The producer needs to interpret the request (the only way to do this is by parsing the request).

This doesn't promote loose coupling, as both consumer and producer are now bound to their implementations. If the producer changes the service internal implementation (without breaking the interface), the solution could potentially break the consumer.

9.1.2 Strong typed service

A strong type service is a service that defines a clear schema when it comes to input and output parameters.

```
Public Employee getPerson(SocSecurityNumber socSecurityNumber)
```

[discuss this further]

9.2 Best Practices

9.2.1 Interfaces should be strong typed

We've seen that the method below really doesn't say a lot. The method takes a String as input (format unknown), and returns the result as a String.

```
public String execute(String request)
```

The method below is a so called strong typed interface. It will always return a person, and will need a Social Security Number as input. Due to the fact that the social security number is strong typed, additional validation can occur on this level. So even before the request is issued, we have a level of validation (only valid social security numbers can be used as input)

```
Public Employee getPerson(SocSecurityNumber socSecurityNumber)
```

9.2.2 Interfaces should have meaningful parameters

As the service interface is part of the public contract, it's important that the parameters that are a part of that interface have meaningful values.

Consider the method below. By looking at the interface, it's clear that we can retrieve an Employee based on its social security number.

```
Public Employee getPerson(SocSecurityNumber socSecurityNumber)
```

Although the method below does exactly the same thing, it isn't as declarative as the method above.

```
Public Employee getPerson(int number)
```

Having meaningful parameters provides an aid for requestors to interpret the interface correctly

9.2.3 Interfaces should be document centric

9.2.4 Interfaces should be compliant with WS-I Basic Profile

9.2.5 Interfaces should be rich enough to shield the user away from implementation changes

9.3 Conclusion

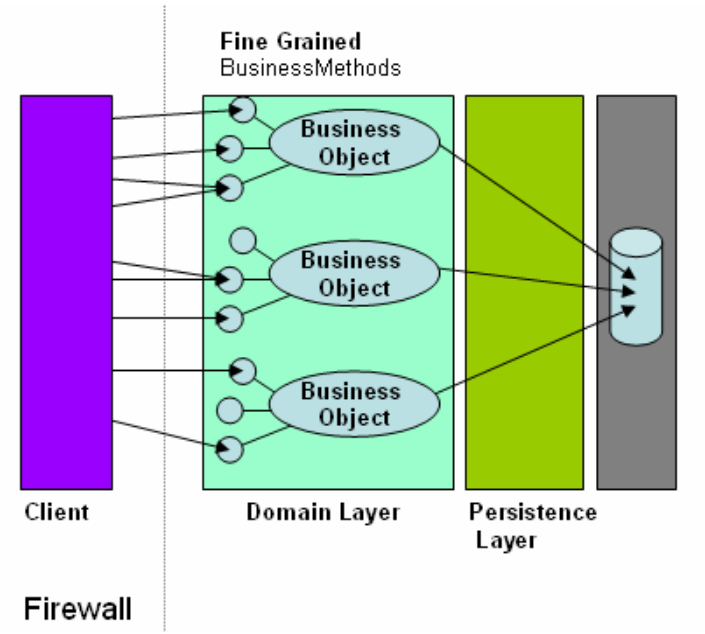
A good interface

- Is strong typed
- Has meaningful parameters
- Is document centric (as opposed to method centric)
- Conforms to WS-I Basic Profile
- Should shield the user away from changes in the implementation

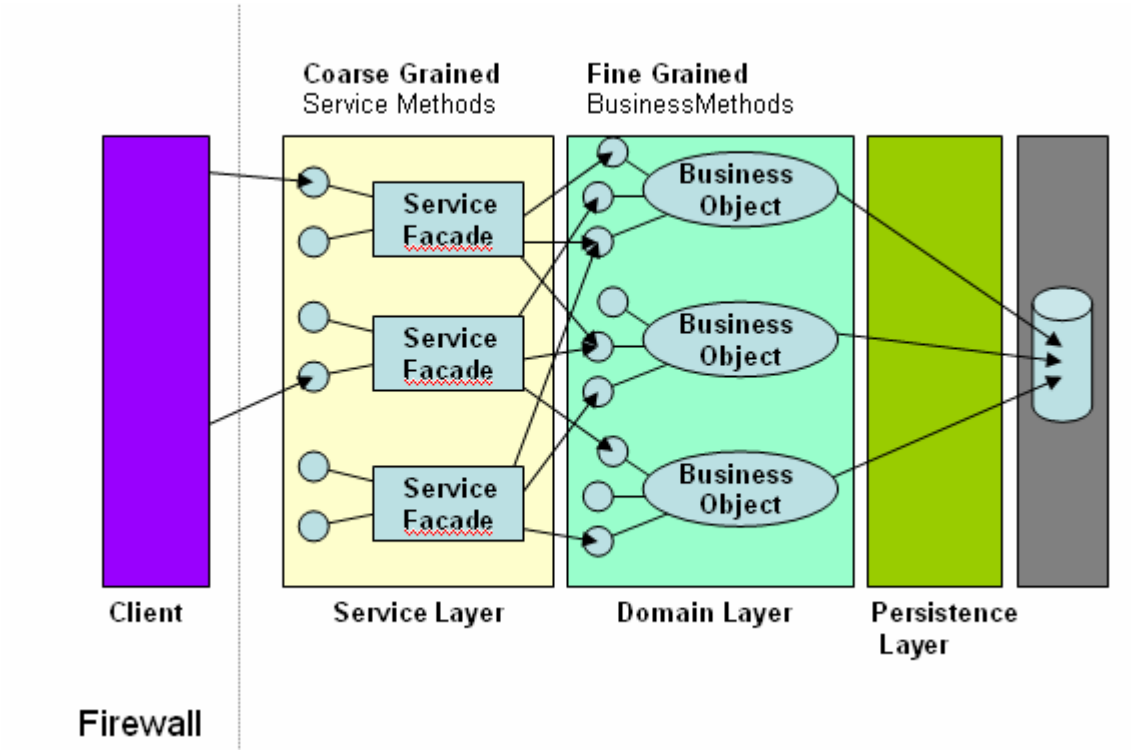
9.4 Service Interface Granularity

Much of the same principles that we have seen in the application architecture section in terms of exposing services internally to the application, come into play when exposing business logic as webservices.

Service interfaces that need to be consumed by web service clients should be course grained
Business objects typically expose very fine grained methods that require a lot of interaction with the client in order to fulfill a certain use-case. This requires us to do a lot of network roundtrips, marshalling data back & forth, and increasing the coupling between the consumer and the producer.



Introducing a set of service facades, that expose course grained operations to the web service clients helps us to reduce the network roundtrips and decreases the coupling between the consumer and producer. The consumer will only needs to perform a single method call, whereas in the previous scenario, the consumer would have to know the various methods calls (and the order in witch to perform them) in order to execute a single use case.

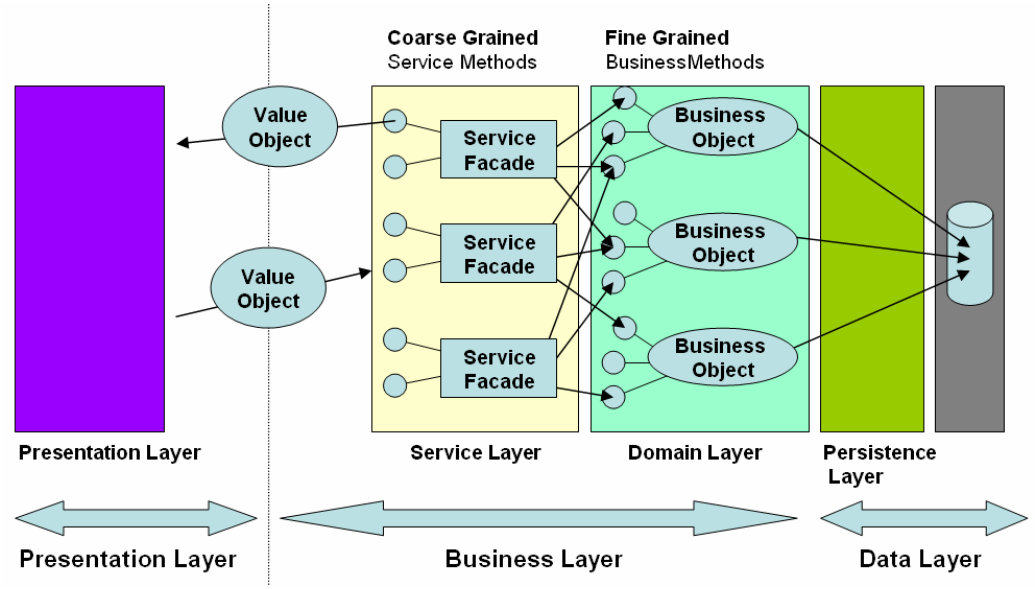


9.4.1 Conclusion

Interfaces that have the right granularity level will be able to

- Perform a business function in a single method call
- Promote reusability
- Model a certain business processes

9.5 Avoiding the use case mismatch



We've already seen that it's a good principle to introduce a service layer in the architecture containing one or more service facades in order to expose business logic.
In the picture above for example, we see how the presentation layer (typically a web application) , can make use of those service facades in order to construct a user interface to the end user.

The same principles go for exposing your application business logic as a web service. One pitfall that should be avoided however is reusing existing service facades (that are already used by the presentation tier of the application).

In order to reduce coupling, it's considered a good principle to separate the service facades used for pure UI services, and those used for web services. Besides the fact that it will help you avoid the functional mismatch (services exposed for the user interface are typically targeted specifically to map to certain screens in the user interface, whereas web services typically require yet another level of exposure.)

10 Service versioning

Anyone who has ever done enterprise development will understand the importance of a solid versioning strategy.
Service versioning is something that isn't really covered by any web service standard. Although a very important aspect during the rollout of a Service Oriented Architecture, no official standard has been published to setup a service versioning scheme. Instead, we have to rely on a set of best practices in order to introduce the concept of versions into the architecture.

Before we look at the different ways of adding versioning, we need to make the distinction between the service implementation, and the service interface. Typically, service implementations are far more flexible when it comes to applying changes to them. A service can evolve easily by applying changes to the implementation, without breaking the contract with existing clients. Service interfaces (defined in WSDL, and published via UDDI) are far more static in nature, and introducing changes on that level will break your existing service contracts, forcing you to adopt some kind of versioning scheme with regards to your web services.

10.1 Using XML Schema for versioning

XML Schema Namespace element

Major version releases should be propagated in the XML Schema Namespace that is associated with the service:

targetNamespace=http://www.mycorp.com/service1/reg/1.1

Changes made to XML Schema namespaces will break existing clients, as they will be unable to marshall/unmarshall the service messages. This means that an action will need to be performed by the consumers. Therefore, this mechanism should only be introduced in order to communicate major service releases that will break the contract between the consumer and producer.

Namespaces in XML Schema are primarily used to define a certain scope within the document (in order to avoid name collisions). However, they can also be used as a mechanism for versioning XML documents.

Using XML Schema version attribute

The XML Schema version attribute also allows for versioning web services. The problem with this approach however is that most validation tools will omit checking the version attribute , so custom code will need to be introduced the interpret the version attribute value.

The upside of this mechanism is the fact that the version attribute (in conjunction with the targetNameSpace attribute), can act as a version control mechanism for small, incremental changes that don't necessarily break the compatibility with the clients (adding a new operation to the WSDL for example).

[Insert example here]

Introducing a custom version attribute

Another option is to include a proprietary version attribute to your schema documents.

A good principle is to use the XML Schema targetNamespace to indicate major versions. It's very easy to implement, and has widespread support.

[Insert example here]

Introduce naming conventions

Having some kind of naming convention when it comes to services can also help you in setting up a version strategy.

You can either opt for

- Major – Minor version
  embedded in the service name (getUser_v1_1)
  embedded in the namepace <xsd:schema targetNamespace=http://www.acme.com/types/pcconfig/v1/1
- Date stamps
  <xsd:schema targetNamespace=http://www.acme.com/2004/03/01/pcconfig,

[elaborate]

10.2 Using UDDI

The UDDI registry can also be used as a way to handle the versioning of webservices.

[describe in detail the relationship between a WSDL document and the UDDI objects]

10.3 Running multiple versions of the same service

In a Service Oriented Architecture multiple versions of the same service can run simultaneously. For example v1.1 of WebServiceX can be backwards compatible with v1.0. Clients who are still using v1.0 can work with either v1.0 or v1.1

This kind of versioning scheme allows for

- A gracefull removal of older version
- The consumers to upgrade when the time is right for them.
- The consumer can be given adequate time to upgrade the client software before the older version of the service is retired from the architecture.

That way, the organization can deal with changes in the architecture. The Enterprise Service Bus (the primary intermediate between a service consumer and service producer) can have a lot of added value on this level.

Offering course grained interfaces
Depending on the granularity of the service interface, a service can be upgraded without any effect on the service consumers. In many cases it will be possible to upgrade the service, or let it evolve in a natural way without breaking the contract between the consumer and the producer.

10.4 Service Release Management

An important aspect of web service versioning is to have a solid release management scheme that can be communicated to all participants of the architecture.

When a new version of a service is in the pipeline, the following steps will need to be executed

1. Develop and test the new version of the service
2. Deploy the new version of the service (using WSDL/UDDI)
3. Notify consumers of the arrival of this new version
4. Optionally, create a pilot project with one consumer.
5. As depicted in the release management plan, keep both the old and new version of the service running simultaneously from a predefined time.
6. Allow for the consumers to upgrade their version.
7. Gracefully remove the older version from the architecture (consumers still using the old version should get an error message indicating that the specific service version is no longer supported.

10.5 Leverage vendor offerings

As versioning is a big issue when it comes to web services, developing a versioning mechanism in house would be unadvisable. Several vendors are offering project in the versioning realm. (Including support for routing, transformation …)
The Enterprise Service Bus also provides a layer that can encapsulate these issues.

11 Service development

11.1 Contract First approach

The contract first approach forces you to write your service interface (or service contract), before actually developing the service. This allows you to build the service in an implementation independent way. Usually, the service contract is then used by development tools in order to generate the necessary server code that will become the service implementation.
This approach will improve the interoperability between services (sometimes writing using different technologies/languages)

11.2 WSDL Predefined extensions.

WSDL Predefined extensions. (Available in WSDL 2.0)

12 Schema handling

So far we've focused on exposing services, and have seen these services as a black box. At the core of each service is a schema that represents the actual data model of the service. The schema should be expressed using XML Schema.

The schema exposes the data model of a particular service. In a service oriented architecture, this data model (in addition to the interface) is the only public part of the 'business logic' of a particular service. The data model is visible in the message payload and is externalized through XML Schema.

Using XML Schema, we achieve the following advantages:

- The schema is externalized , standardized and federated
- The schema provides a visual specification targeting developers
- The schema provides visible contract for clients wishing to access a particular service
- XML Schema has excellent tooling support to allow non-IT professionals to interact with them.

The problem however is not so much the versioning of the meta-data in particular, but rather the evolution of the metadata (or data model) in general.

- The data model should be expressed in XML Schema
- The data model should avoid duplication, and should uphold high standards in terms of semantics.
- The data model should be stored in a central repository that allows versioning and reporting.
- Change management should be built in to the system.
- The data model should be used to generate XML schemas that will be applied on the service interface level.
- Managing the data model should be possible in a multi user environment.
- Several (both commercial and non commercial) tools are available that specifically target the need for schema management.
- Model driven architecture (discuss)

[Discuss this further]

13 Securing Web Services

The current Web Services landscape in terms of security is an ongoing process.
Although several standards are starting to emerge, we have yet to seen mature implementations of those standards by the major vendors.
The lack of standards on the one hand, and the level of maturity of the implementations on the other, makes it somewhat difficult to define a clear roadmap when it comes to defining a strategy for securing web services.
This doesn't mean however that we should completely ignore these standards.

13.1 Transport Layer Security

We've already seen that web services are dependant on an underlying transport protocol (HTTP, SMTP…). In order to achieve some basic level of security, it's imperative that the transport protocol also offers some level of security.

We'll discuss the HTTP, HTTPS and SMTP protocol in terms of security.

13.1.1 SMTP

Natively, mailing protocols like POP or SMTP provide very little or no security.
In order to secure the SMTP transport layer, we usually rely on some kind of commercially available product.
A mail system such as Microsoft Exchange Server does provide a sufficient level of security (authentication, encryption, signing…)

13.1.2 HTTP

HTTP, being the most popular protocol used to exchange information on the internet is also natively insecure.
Information is sent as clear text over an unsecured network.

13.1.3 HTTPS

HTTPS, or HTTP via Secure Socket Layer (SSL) allows client side and server side authentication through the use of certificates.
It's generally assumed that HTTPS provides an adequate level of security on the transport level.

HTTPS provides

- Party identification
- Party authentication
- Message integrity
- Message confidentiality

HTTPS doesn't provide

- Authorization
- Auditing

13.2 Service Description Layer Security

This type of security is located at the actual message level. IT basically means that we embed security information inside the XML document that is sent across the wire.

The current web services landscape allows for the following possibilities when it comes to securing XML messages

- XML Digital Signatures
- XML Encryption
- Security Assertion Markup Language (SAML)

13.2.1 XML Digital Signatures

Standard used to verify the origin of an XML message using a variety of different signature algorithms.

13.2.2 XML Encryption

XML Encryption is an emerging standard for encrypting either an entire, or individual parts of an XML document.

13.2.3 Security Assertion Markup Language (SAML)

SAML is a product of the OASIS Security Services Technical Committee
It's an XML standard for exchanging authentication and authorization data between security domains.

13.3 Web Service Security Standards

13.3.1 WS-SecureConversation

WS-SecureConversation describes how to manage and authenticate message exchanges between different parties, built on the concept of trust based on security tokens, including security context exchange and establishing and deriving session keys.

13.3.2 WS-Federation

WS-Federation describes how to manage and broker the trust relationships in a heterogeneous federated environment, including support for federated identities. The concept is known as federated identity management.

13.3.3 WS-Authorization

WS-Autorization describes how access policies for a Web service are specified and managed. In particular, it describes how claims may be specified within security tokens and how these claims will be interpreted at the endpoint.

13.3.4 WS-Policy

WS-Policy describes the capabilities and constraints of the security (and other business) policies on intermediaries and endpoints (for example, required security tokens, supported encryption algorithms, and privacy rules).

13.3.5 WS-Trust

WS-Trust describes a framework for trust models that enables Web services to securely inter operate, by defining a set of interfaces that a secure token service may provide for the issuance, exchange, and validation of security tokens.

13.3.6 WS-Privacy

WS-Privacy describes a model for how Web services and requestors state privacy preferences and organizational privacy practice statements.

14 Interoperability

Interoperability is without a doubt one of the most important design principles when rolling out a service oriented architecture. Web services are an ideal mechanism to achieve this, but the story doesn't end there. By using standards such as SOAP, WSDL, UDDI, we only have one piece of the puzzle.
In order to offer true interoperability between vendors/technologies that will potentially make use of the architecture, the service oriented architecture should be compliant with the WS-I Basic Profile. WS-I (Web Service Interoperability Organization) is an open, industry organization committed to promoting interoperability among web services based on common, industry-accepted definitions and related XML standards support.

Interoperability is addressed through profiles, and currently, the Basic Profile (the first profile coming out of the WS-I organization) is the primary building block for achieving interoperability.

WS-I Basic Profile includes

- XML Schema 1.0
- SOAP 1.1
- WSDL 1.1
- UDDI 2.0

Web Service platforms (SOAP toolkits, Web Service products , ….) that are compliant with WS-I Basic Profile ensure that applications / services running on those platforms are portable across the various technological boundaries that we can find within a SOA.

[discuss this further]