Business Process Digitalization and Cloud Computing

7. Design Service Interface

Andrea Morichetta, Phd

November 29, 2017

Computer Science Division

Table of contents

Services

- A service provides **capabilities** that are accessed through its interface.
- The interface describes how those **capabilities are presented** and the **rules** and **protocols** for using them.

A service is a combination of:

- its **interface**: the public view of the service,
- its **implementation**: the private view of the service

The service interface:

- **hides the details** of the implementation.
- expresses the services **functions** (operations) that it provides.
- provide the **schema of the information** (parameters of the service operations) derived from a common semantic model.

Service Characteristics

TYPE	VISIBILITY	SCOPE	GRANULARITY	RESPONSIBILITY
Business Service	Public	Enterprise	Medium to Large	Implements discrete business function across lines of business.
	Public	Line-of-Business (LOB)	Medium to Large	Implements discrete business function. Represents a logical group of related functions.
Domain Service	Private	Domain	Small to Medium	A domain-specific subunit of processing. General purpose, reusable.
Utility Service	Public	Enterprise	Small to Medium	A common subunit of processing. Supports semantic business objects. Applicable across multiple LOBs.
Integration Service	Private	Solution	Small to Large	Exposes business operations. Groups related transactions. Provides a single point of contact.

Service Integration

Interaction Style

Interaction style describes the pattern of the service's operation signature, specifically how the **information is passed into and out of the service.** The interaction style is described in a Message Exchange Pattern (MEP).

There are two separate concerns of MEP:

- how the **information is passed**
- **type of message** and **synchronization**.

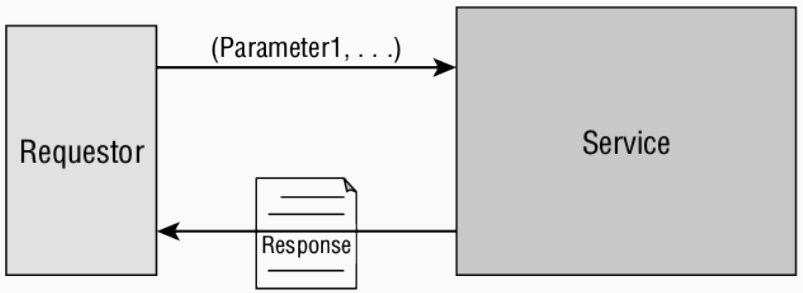
How the information is passed

Parameter Passing

- The operation signature contains **one or more individual parameters**.
  Example:

  ```
  Response = service operation (param1, param2, ...);
  ```

- The **inputs are passed as parameters**, which are typed by standard or custom data types.
- The **output** can be simple type or a complex type.
- **Works well** with **well-defined and constrained** inputs and with **small granularity services**



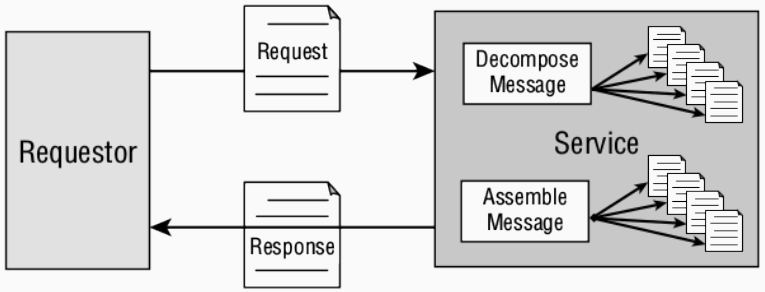
How the information is passed

Document Passing

- The operation signature contains one **request** (input) **document** and one **response** (output) **document**.
  Example:

  ```
  Response document = service operation (request doc.);
  ```

- The inputs and outputs are passed as documents.
- **Works well** with larger granularity services where the input and output may contain a lot of data



- Service might decompose the request into individual documents that are passed separately to the services

How the information is passed

Data Passing

- The operation signature contains one or more **request parameters** and one **response** (output) **document** or **dataset**

  Example:

  ```
  Response data = get operation (entityID);
  ```

- The service usually supports an operation that **returns the entire data set** about a specific entity ID.
- common variation on the data-passing interface allows the requestor to pass in a **specific query** or to ask for a customized subset of information

Invocation Style

Request/Reply Synchronous case

- The consumer **sends a request** to a service and **waits**.
- When the service has **processed** the request, it **sends a reply**.
- The consumer receives the reply and **resumes processing.**

Request/Reply Asynchronous case (store-and-forward)

- The consumer **sends an asynchronous request** to a service and **continue to processing other tasks**.
- Some times later the consumer **look for the reply** and then **processes it**. (e.g. email)
- **Issues**:
  - no answer
  - correlation between request and response (Identifier)

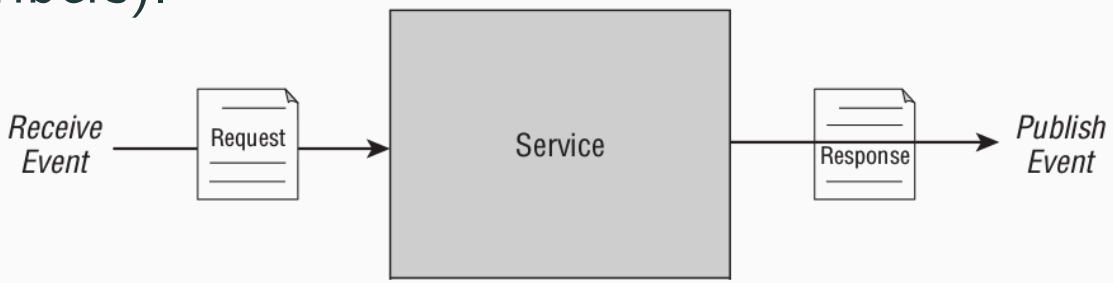
Invocation Style

One-way messages

- The consumer send a message to a service but **don't expect any response**
- One-way message is often associated with a **guaranteed delivery messaging infrastructure.**

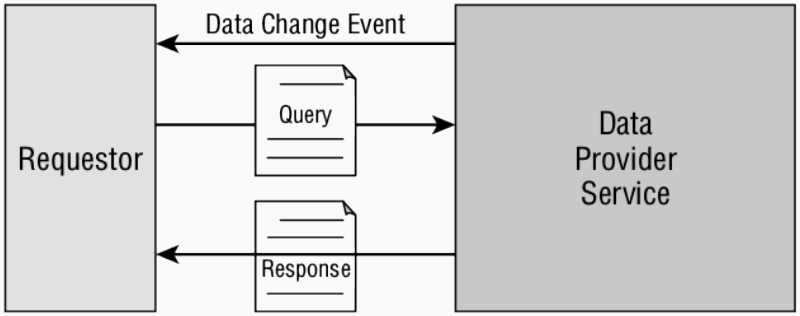
Events

- Event-driven architectures **rely on an intermediary**, or an "event broker" to **receive notification** from event sources (publish) and **inform (invoke) all interested parties** (subscribers).



Mixed Style

- Mixed paradigm is given by a **combination of styles**.
- The **data provider publishes data change events**
- Consumers that are interested in data changes **subscribe to these events**, most consumer are subscribed with only a subset of all the data changes and ignore most events
- When the event is about data they are interested, the consumer (subscriber) then makes a request/response invocation to the data provider to get the specific data that they care about.



Stateless Interface

- Service interface should be stateless as possible.
- Service **does not maintain state** on behalf of its consumer between requests.
- State:
  - **Execution state**: represents the **state of the service** during its execution. It includes internal variables created during service execution for keeping track of partial results during the execution or storing parameters between multiple components of a service implementation
  - **Invocation state**: is a shared context between the service consumer and service provider.
    A service may participate in multiple conversations and keep track of each conversation separately

Design model interface

Design model

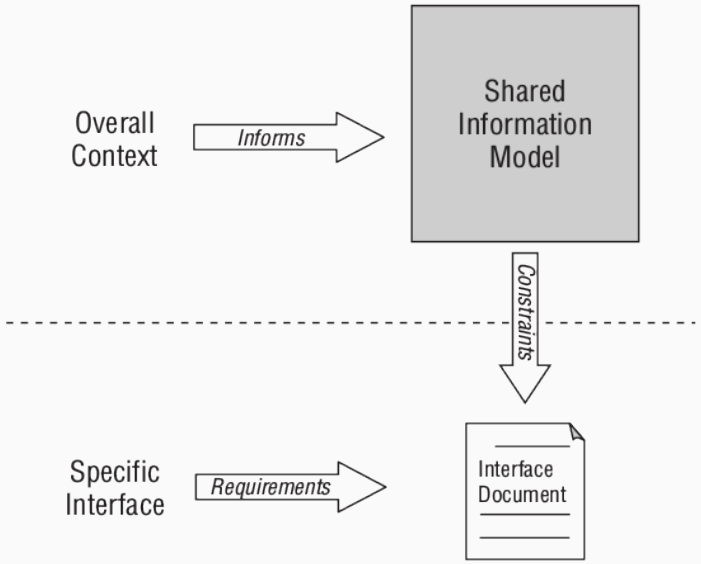
To summarize the **information model** answer the following questions:

- What **information is shared** between services?
- What **information must be passed into** and out of each service?
- What **information needs to be common** across services?

One goal of the information model is to identify information that must be common and shared among services. It follows then that the **information passed through the service interface should conform to the information model**.

Document-passing interface

Interface document contains the shared information that is passed between services, and that the definition of that document is described in a document schema.



The schema must be based on the shared information model (in fact, it is a subset of the model).

Every element of the schema must exist in the information model.

WSDL

WSDL Document

An WSDL document describes a web service. It specifies the **location of the service, and the methods of the service**, using these major elements:

Element	Description
<types>	Defines the (XML Schema) data types used by the web service
<message>	Defines the data elements for each operation
<portType>	Describes the operations that can be performed and the messages involved.
<binding>	Defines the protocol and data format for each port type

How the WSDL look like?

```
<definitions>

<types>
data type definitions........
</types>

<message>
definition of the data being communicated....
</message>

<portType>
set of operations......
</portType>

<binding>
protocol and data format specification....
</binding>

</definitions>
```

WSDL example

```
<message name="getTermRequest">
   <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
      <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
   <operation name="getTerm">
      <input message="getTermRequest"/>
      <output message="getTermResponse"/>
   </operation>
</portType>
```

The <portType> Element

The <portType> element defines:

- a web service
- the operations that can be performed
- the messages that are involved

Type	Definition
One-way	The operation can receive a message but will not return a response
Request-response	The operation can receive a request and will return a response
Solicit-response	The operation can send a request and will wait for a response
Notification	The operation can send a message but will not wait for a response

One-way Operation

```
<message name="newTermValues">
  <part name="term" type="xs:string"/>
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="setTerm">
    <input name="newTerm" message="newTermValues"/>
  </operation>
</portType >
```

WSDL Request-Response Operation

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

WSDL Binding to SOAP

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
</operation>
</portType>

<binding type="glossaryTerms" name="b1">
  <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http" />
     <operation>
       <soap:operation soapAction="http://example.com/getTerm"/>
         <input><soap:body use="literal"/></input>
         <output><soap:body use="literal"/></output>
     </operation>
</binding>
```

WSDL Binding to SOAP

The binding element has two attributes:

- **Name**: defines the name of the binding
- **Type**: is an attribute that points the port for the binding

The soap:binding element has two attributes:

- **Style**: can be "rpc" or "document".
- **Transport**: defines the SOAP protocol to use.

The operation element in the binding defines each **operation that the portType exposes.**

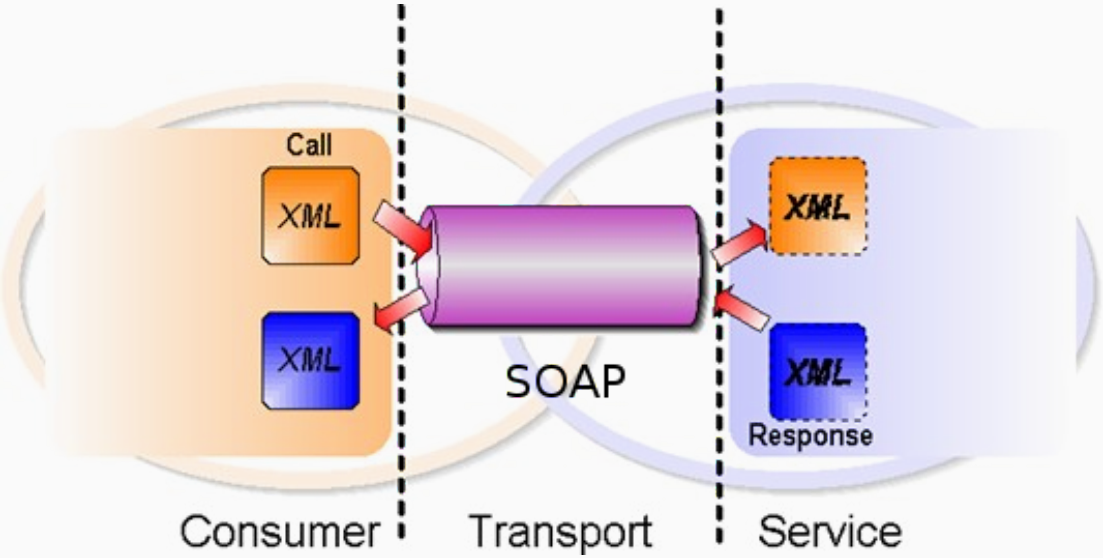
- For each operation the corresponding SOAP action has to be defined.
- It is al necessary to specify how the input and output are encoded using the *use* keyword.

SOAP

Why SOAP?

SOAP provides a **way to communicate between applications** running on different operating systems, with different technologies and programming languages.

`https://www.w3.org/TR/soap12/`



The communications are encoded with soap but **transported by HTTP.**

Beyond Client-Server

- SOAP assumes messages have an **originator**, one or more ultimate **receivers**, and zero or more **intermediaries**.
- The reason is to support distributed message processing.
- Implementing this message routing is out of scope for SOAP.
- Assume each node is a Tomcat server or JMS broker. That is, we can go beyond client-server messaging.

SOAP Key factors

- SOAP is just a message format.
  - Must transport with HTTP, TCP, etc.
- SOAP is independent of but can be connected to WSDL.
- SOAP provides rules for processing the message as it passes through multiple steps.
- SOAP payloads
  - SOAP carries arbitrary XML payloads as a body.
  - SOAP headers contain any additional information
  - These are encoded using optional conventions

Web services messaging requirements?

- Define a message format
  - Define a messaging XML schema
  - Allow the message to contain arbitrary XML from other schemas.
- Keep It Simple and Extensible
  - Messages may require advanced features like **security, reliability, conversational state**, etc.
- Tell the message originator **is something goes wrong**.
- Define data encodings
  - the message recipient should be informed about the types of each piece of data.

Web services messaging requirements

- Define some RPC conventions that match WSDL
- **Decide how to transport the message.**
  - Generalize it, since messages may pass through many entities.
- Decide what to do about **non-XML payloads** (movies, images, arbitrary documents).

SOAP Structure

A SOAP message is an ordinary XML document containing the following elements:

- An **Envelope** element that identifies the XML document as a SOAP message
- A **Header** element that contains header information
- A **Body** element that contains call and response information
- A **Fault** element containing errors and status information

The SOAP Envelope Element

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  ...
  Message information goes here
  ...
</soap:Envelope>
```

- The envelop is the **root container** of the SOAP message
- The **namespace** defines the Envelope as a SOAP Envelope. If a different **namespace** is used, the application generates an error and discards the message.
- The **encodingStyle attribute** is used to define the data types used in the document.

The SOAP Header Element

```
<?xml version="1.0"?>
<soap:Envelope.....>

<soap:Header>
   <m:Trans xmlns:m="http://www.w3schools.com/transaction/"
   soap:mustUnderstand="1">234
   </m:Trans>
</soap:Header>
...
...
</soap:Envelope>
```

- The SOAP Header element contains **application-specific information** (like authentication, payment, etc) about the SOAP message.
- The attributes defined in the SOAP Header defines how a **recipient should process the SOAP message**.

Header Possible attributes

- The **mustUnderstand** attribute can be used to indicate whether a header entry is **mandatory or optional** for the recipient to process.
  - If it is =1 the receiver processing the Header must recognize the element otherwise it will fail.
- The **actor** attribute is used to address the Header element to a specific endpoint.
- The **encodingStyle** attribute is used to define the data types used in the document.

The SOAP Body Element

The **Body element** contains the actual **SOAP message** intended for the ultimate endpoint of the message.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body>
   <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
      <m:Item>Apples</m:Item>
   </m:GetPrice>
</soap:Body>

</soap:Envelope>
```

- The example above requests the price of apples

The SOAP Body Element Response

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body>
   <m:GetPriceResponse xmlns:m="http://www.w3schools.com/prices">
      <m:Price>1.90</m:Price>
   </m:GetPriceResponse>
</soap:Body>

</soap:Envelope>
```

The SOAP Fault Element

The optional SOAP Fault element is used to indicate error messages.

Sub Element	Description
faultcode	A **code** for identifying the fault
faultstring	A **human readable explanation of the fault**
faultactor	Information about **who caused the fault** to happen
detail	Holds application specific error information related to the Body element

SOAP Fault Codes

Error	Description
VersionMismatch	Found an invalid namespace for the SOAP Envelope element
MustUnderstand	An immediate child element of the Header element, with the mustUnderstand attribute set to "1", was not understood
Client	The message was incorrectly formed or contained incorrect information
Server	There was a problem with the server so the message could not proceed

The HTTP Protocol

A HTTP **request message** send to the server:

```
POST /item HTTP/1.1
Host: 189.123.255.239
Content-Type: text/plain
Content-Length: 200
```

The server sends an HTTP **response** back to the client:

```
200 OK
Content-Type: text/plain
Content-Length: 200
```

Server could **not decode** the request:

```
400 Bad Request
Content-Length: 0
```

EXAMPLE

Web Server WSDL

SOAP test UI

WSDL Validator

UDDI

UDDI

- UDDI stands for **Universal Description, Discovery, and Integration**
- Represent a technical specification for **publishing and finding** businesses and Web services
- UDDI 1.0 was originally announced by Microsoft, IBM and Ariba in September 2000
- In May 2001, Microsoft and IBM launched the first UDDI operator sites
- UDDI 2.0 was announced in June 2001
- Approved by the Organization for the Advancement of Structured Information Standards (OASIS) as a formal standard in April 2003
- Currently UDDI 3.0 has been published as OASIS committee specifications

UDDI Goals

Consist of two parts:

- A technical specification for **building distributed directory** of businesses and web services
- **business registry**: is a sophisticated naming and directory services.

UDDI defines **data structures** and **APIs** for publishing service descriptions in the registry and for querying the registry to look for published descriptions.

Goals of UDDI

Two main goals with respect to service discovery:

- to support developers in **finding information** about services
- to enable **dynamic binding**, by allowing clients to query the registry and obtain references to services of interest

The data captured within UDDI is divided into three main categories:

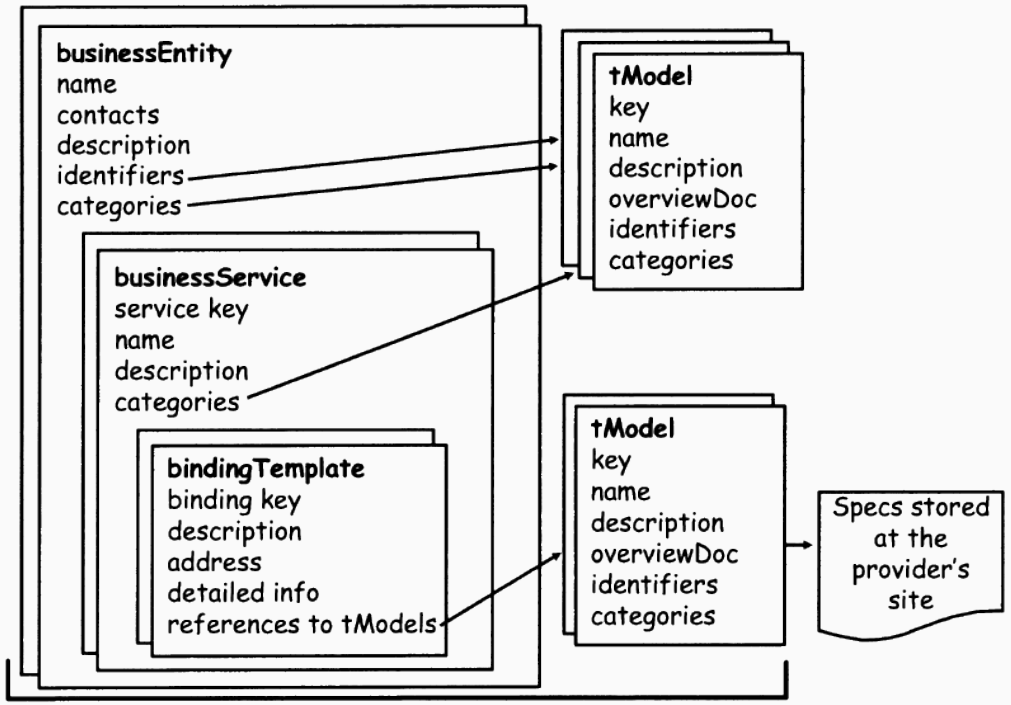
- **White pages**:
  - Contain general info about a specific company
  - E.g. Business name, business description, contact info, address and phone numbers.

Information in UDDI

- **Yellow pages**
  - Extend the ability to locate a business or service.
  - Support **classification** using various **taxonomy** systems for categorization
- **Green pages**
  - Provide information on how and where to programmatically invoke a service
  - Contain technical info about a Web service
  - Provide address for invoking service
  - Not necessary SOAP-based service
  - Can provide references to a Web page, email address or services using other component technologies, CORBA, RMI, etc.

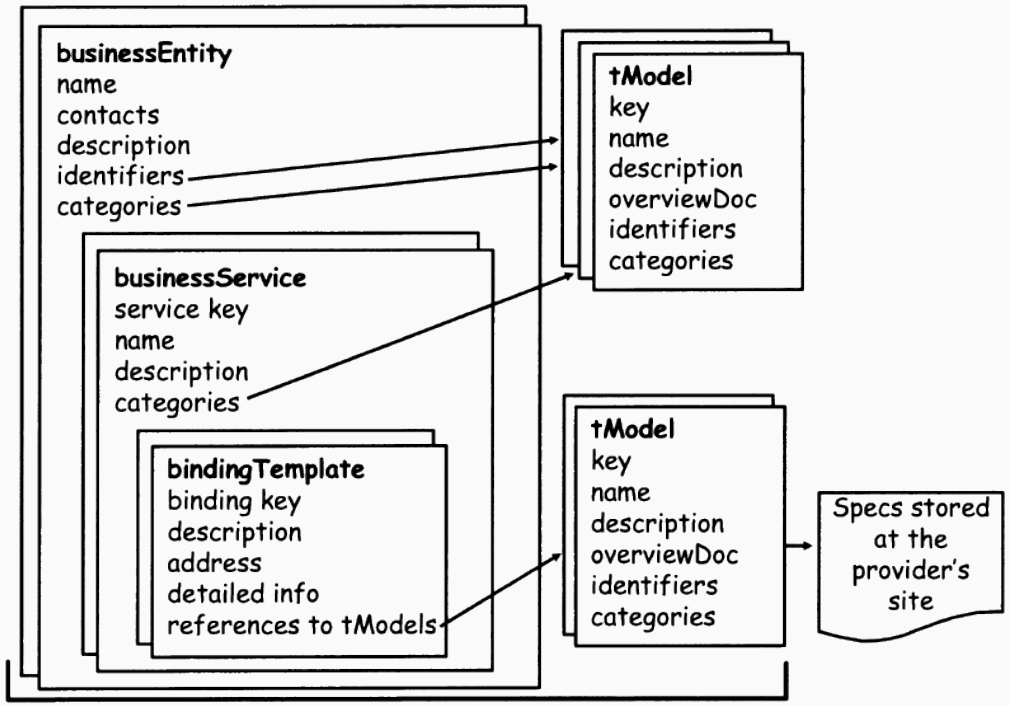
UDDI Data structures

- **Businessinformation**: Describe the organization that provides web services.
- **BusinessService**: Typically correspond to one kind of services, but provided at different addresses, in multiple versions, and through different technologies



UDDI Data structures

- **bindingTemplate**: This element describes the technical information necessary to use a particular Web service(address, tModel).
- **tModel**: Is a technical model and is a container for any kind of specification (service interface, interaction protocol).



UDDI API

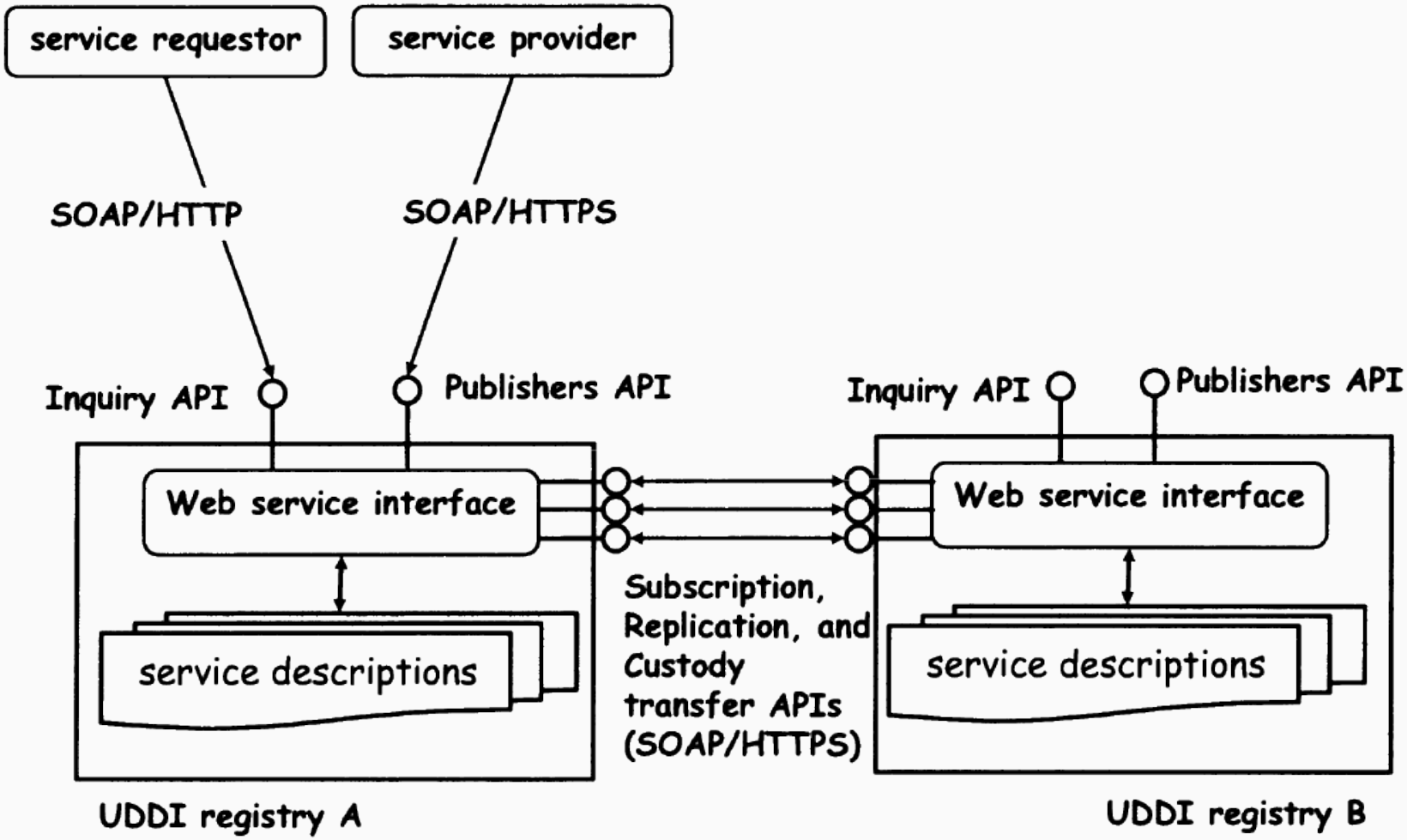
- There are two ways to search or publish a business/service
  - **Using the Web pages** provided by the UDDI implementation (UDDI cloud service), such as uddi.microsoft.com (Need human intervention)
  - **Using the APIs** provided by UDDI
    Can be made automatic by calling the APIs with computer programs

UDDI API

UDDI APIs can be divided into three parts:

- Inquiry APIs: includes **operations to find registry entries** that satisfy search criteria and to get overview information about those entities. (*find_business*, *find_service*, *find_binding*, *find_tModel*)
- Publishing APIs: permit the service providers to add, modify, delete entries in the registry. Operations are (*save_business*, *save_service*, *save_binding*, *save_tModel*)
- Security APIs: used to get and discard authentication tokens (*get_authToken*, *discard_authToken*).

Interaction between UDDI registry



Example: find_business request

```
<envelope xmlns=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <body>
   ------------------------------------------------
   | <find_business generic="1.0" xmlns=urn:uddi-org:api>
   | <name>XMethods</name>
   | </find_business>
   ------------------------------------------------
   </body>
</envelope>
```

- UDDI api find-business
- look for Xmethod

Example: find_business response

```
<envelope xmlns=
   http://schemas.xmlsoap.org/soap/envelope/>
   <body>
    <businessList generic=1.0 operator=Microsoft
     Corporation truncated=false xmlns=urn:uddi-org:api>
      <businessInfos>
      ------------------------------------------------
      | <businessInfo businessKey=
      | ba744ed0-3aaf-11d5-80dc-002035229c64>
      | <name>XMethods</name>
      | <description> ... </description>
      | <serviceInfos>
      | <serviceInfo> ... </serviceInfo>
      | </serviceInfos>
      | </businessInfo>
      ------------------------------------------------
         </businessInfos>
       </businessList>
     </body>
</envelope>
```

Example: get_businessDetail request

```
<envelope xmlns=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <body>
    ----------------------------------------
    | <get_businessDetail generic=1.0
    | xmlns=urn:uddi-org:api>
    | <businessKey>
    | ba744ed0-3aaf-11d5-80dc-002035229c64
    | </businessKey>
    | </get_businessDetail>
    ----------------------------------------
   </body>
</envelope>
```

- Query for a businessEntity record based on its key

Example: get_businessDetail response

```
<envelope xmlns=
   http://schemas.xmlsoap.org/soap/envelope/>
  <body>
    <businessDetail generic=1.0 operator=Microsoft Corporation
     truncated=false xmlns=urn:uddi-org:api>
     ---------------------------------------
     | <businessEntity businessKey=
     | ba744ed0-3aaf-11d5-80dc-002035229c64>
     | <name>XMethods</name>
     | <description> ... </description>
     | <contacts>
     | <contact> ... </contact>
     | </contacts>
     | <businessServices>
     | :
     | </businessServices>
     | </businessEntity>
     ---------------------------------------
    </businessDetail>
  </body>
</envelope>
```

Pubblication of a process

- The objective of publishing business is to allow the clients to know the details of the business
  - such as the name of the company, the contact person, address and phone number etc.
- By publishing the service, the clients would know where and how to contact the service provider
  - such as the access point (or URL) of the service, transport protocol used (HTTP, FTP or else)
- Publishing the tModel allows the clients to invoke the service provided by the business
  - based on the WSDL document of the service

Questions?