Università degli Studi di Bologna
Facoltà di Ingegneria

Principles, Models, and Applications for Distributed Systems M

WEB SERVICES implementation and usage

Luca Foschini

Web Services

Differences between services offered on Web and Web Services

Users can **use services offered on a Web site** using an integrated system, **C2B**

Web Services (WS), instead, are standard to obtain via web the mechanisms offered by a programming language typically **B2B**

Based on HTML-compatible environments; in addition, we assume to use tools that consider more recent and available extensions such as XML (eXtensible Markup Language)

Open environment perspective

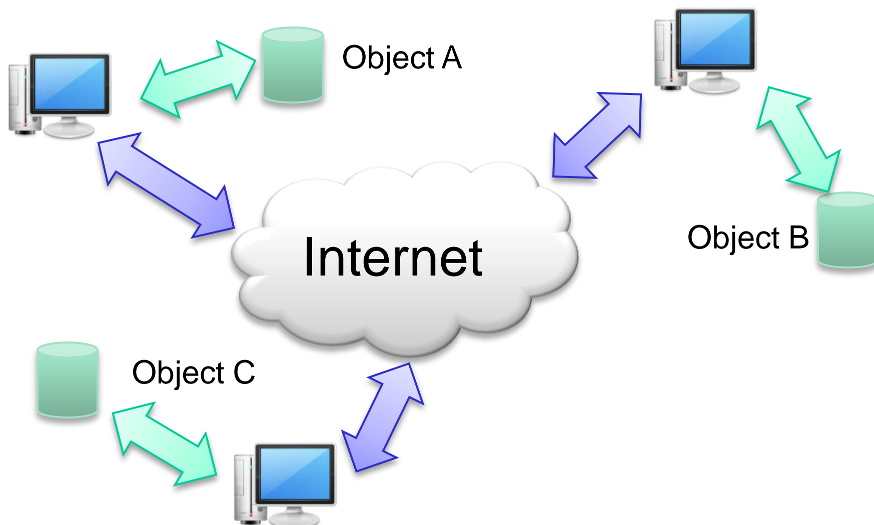
MIDDLEWARE for system support

MIDDLEWARE AND COMPONENTS:

state-of-the-art and future directions

Providing services for distributed, pervasive, ubiquitous computing

Services as systems or frameworks for *integration and composition* of distributed objects.



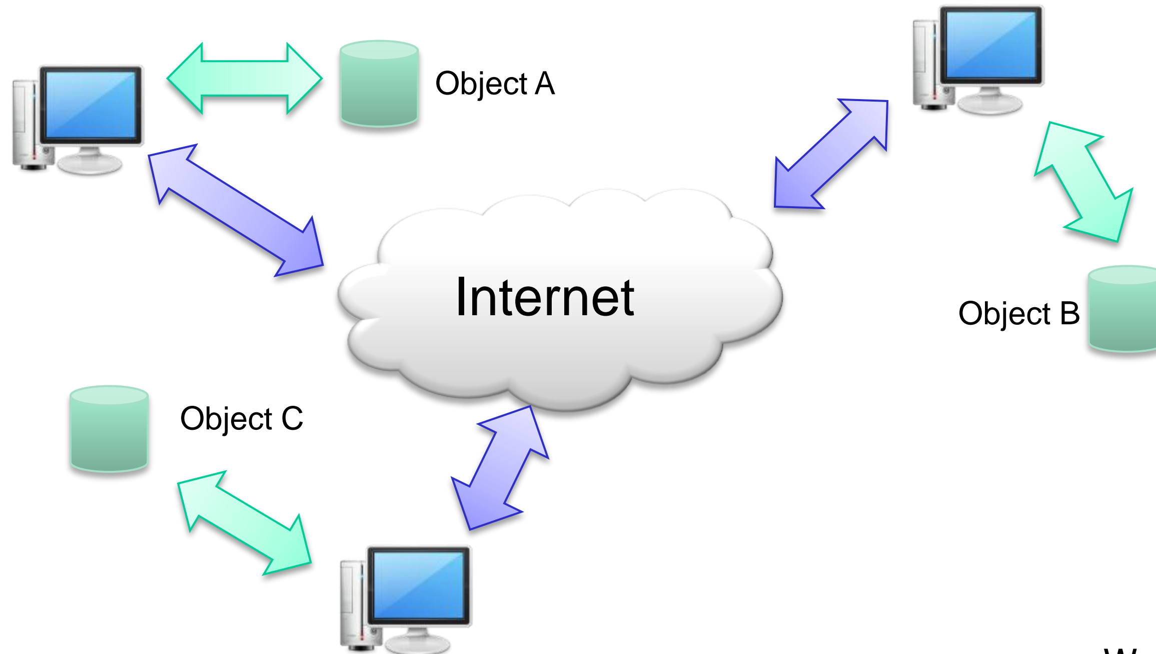
With **portability across heterogeneous systems** and **security checks** guarantees

Object-oriented MIDDLEWARE

Widely deployed MIDDLEWARES based on a Client-Server architecture: RPC (C) and RMI (Java)

Problems:

- **Dependant on a programming language**
- **Hard to integrate with existing (possibly legacy) systems and tools**



Web Services as protocols and standards

Web Services as Integration MIDDLEWARE

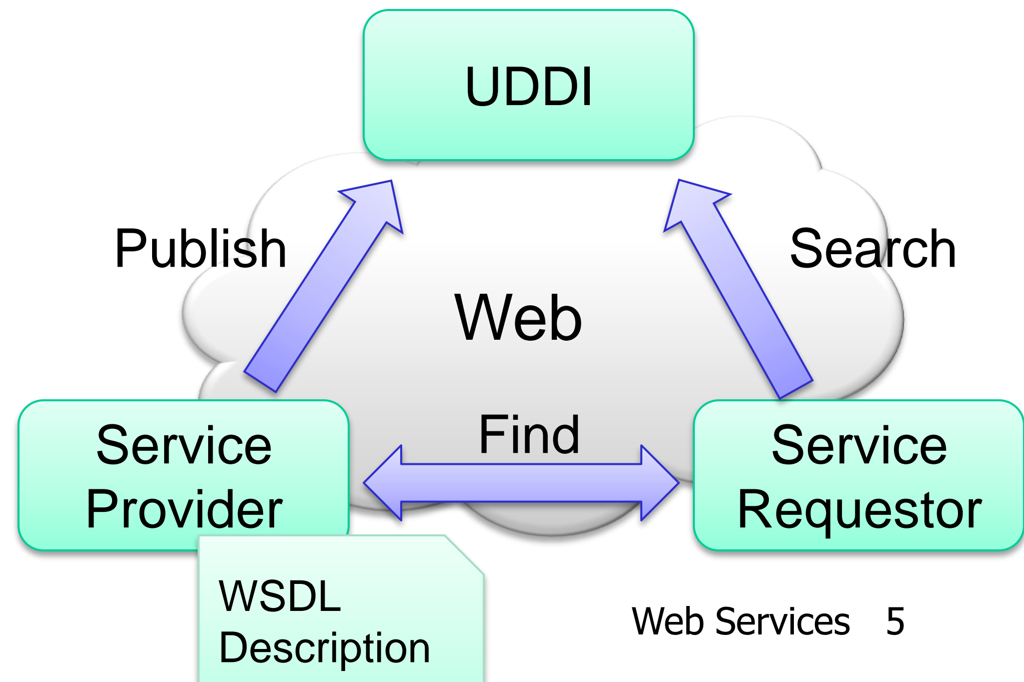
SOAP (Simple Object Access Protocol)

WSDL (Web Services Description Language)

UDDI (Universal Discovery, Description and Integration)

and other extensions

To enable interoperability using programming over Web (exploiting XML)



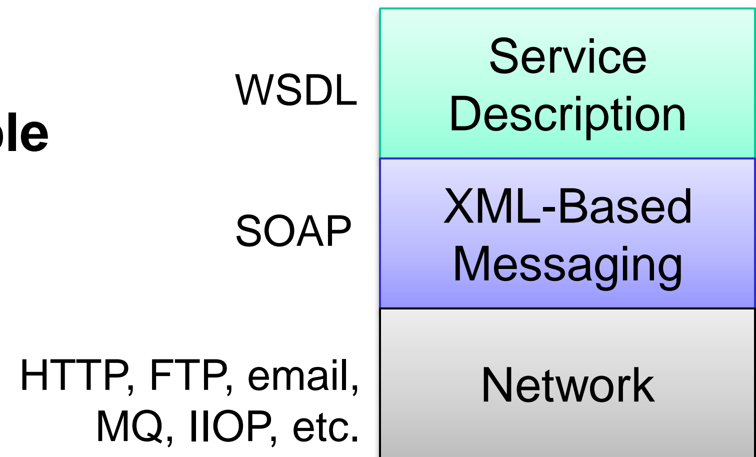
Web Services: Protocols

SOAP

Communication protocol for C/S interaction, for both requests and responses

WSDL

XML-based language to describe available services



UDDI

Name system to import and export the properties of services

there are also other extensions.

XML – advantages

XML allows to impose structure (not meaning) over typically unstructured data.

XML is compatible with **HTML**, even for already **existing documents.**

XML allows to omit **information structure (if it exists and it is known).**

XML allows to use external tools for **data validation, elaboration, and management.**

XML allows to use **wrapping** to refer to **repeating structures.**

XML **is the de facto** **standard for** **Web Services** **for generalized use.**

SOAP (SIMPLE OBJECT ACCESS PROTOCOL)

SOAP protocol designed to work over **Web protocols** while supporting the **specification**, **design** and **management** of **components** and **operations**.

Solution to support **parameters and values as message payload** and for **remote invocation of objects** based on Web technologies

PROJECT ASSUMPTIONS AND GUIDELINES

- **XML to serialize data**
- **HTTP as transport protocol**

Example

```
<SOAP-ENV:Envelope>
 <SOAP-ENV:Body>
   <m:GetLastTradePrice>
   <symbol>MOT</symbol>
   </m:GetLastTradePrice>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP (SIMPLE OBJECT ACCESS PROTOCOL)

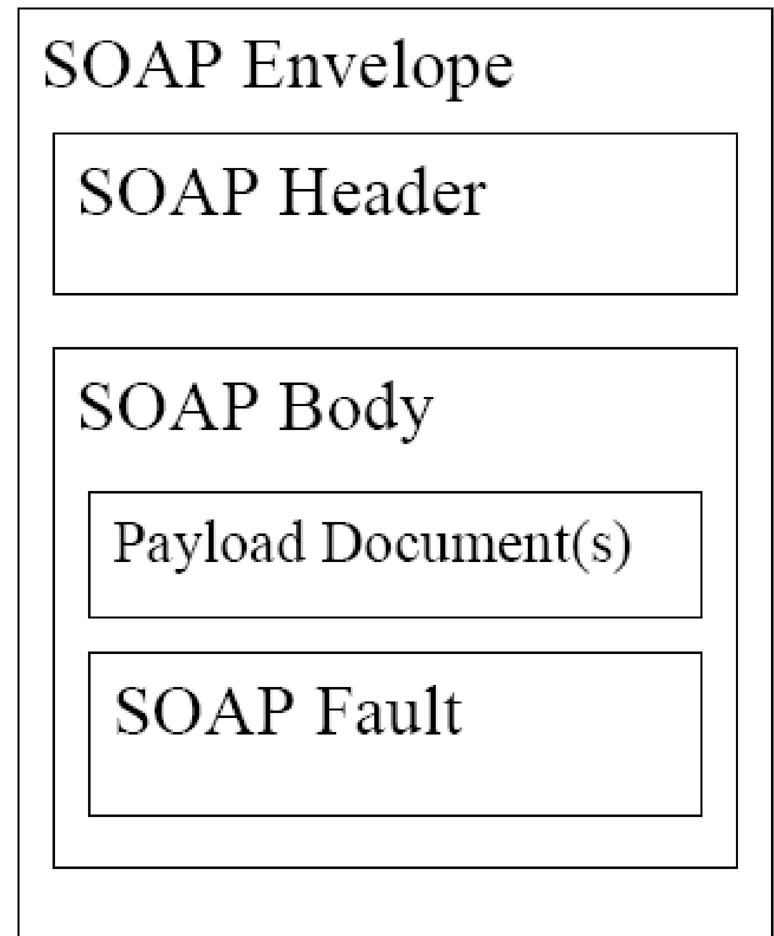
SOAP Protocol

Envelope wraps the **message content**

Header contains **additional informations** (such as security tokens)

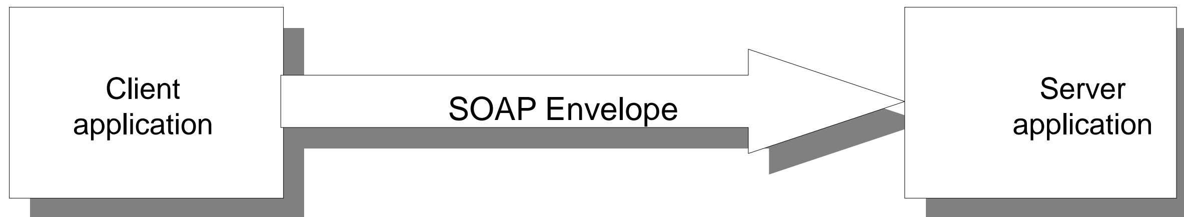
Body wraps **requests and responses** (typically, the message to send)

Fault wraps possible **errors and exceptions**



SOAP (SIMPLE OBJECT ACCESS PROTOCOL)

Typical C/S interaction (between sender and receiver), but with **high interoperability**



Envelope

Header

....

....

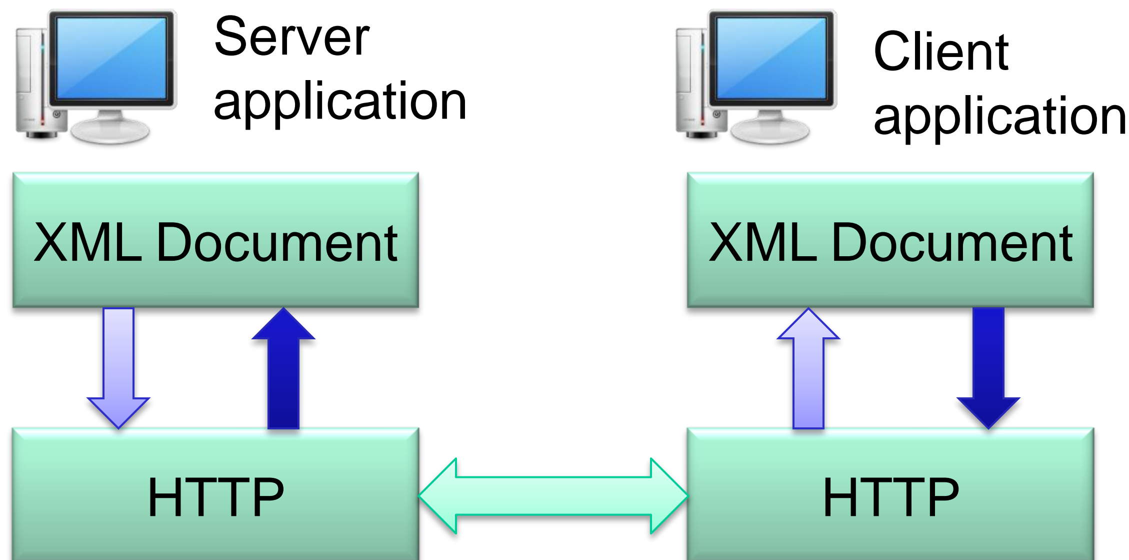
Body

<SOAP-ENV:Body
<m:GetLastTradePrice xmlns:m="some -URI">
<m:symbol>IBM</m:symbol>
</m:GetLastTradePrice>
</SOAP-ENV:Body>

SOAP (SIMPLE OBJECT ACCESS PROTOCOL)

Protocol to send data:

- Platform independent **data serialization**
- **Lightweight, resilient, flexible** operations
- Support for **almost all architectures**
  (**.NET, J2EE, IBM WebSphere, Sun ONE**)



SOAP (SIMPLE OBJECT ACCESS PROTOCOL)

SOAP protocol specifies:

- **interaction style**
  - **document (one-way interaction)**
  - **RPC like**
- **XML elements management**
- **transport**

It DOES NOT specify local interaction

SOAP configures

a stateless interaction protocol

Without providing support to semantic informations of the interaction contract

SOAP typically exploits web operations *GET* and *POST*

SOAP and EXECUTION: example

A simple example: a financial application (client) uses a service that provides real-time stock quotes.

This interaction involves the request of the latest quote of a capital stock and the response from server.

Main steps:

- Client application builds a request in XML format using the SOAP syntax
- Client application sends the request to a **web server** via HTTP
- Server receives and parses the request, transforms it to a command, dispatches it to an application running on the server side
  - ***The application receives the command and retrieves from its database the requested data*** *(as an example)*
- The application **builds a response** in XML format and **returns it to the Web server**
- The Web server returns the result to the client **as an HTTP response**

SOAP and XML (request)

```
<POST /StockQuote/HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset=”utf-8”
Content-Length: nnnn
SOAPAction: “Some-URI”

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope/”
SOAP_ENV:encodingStyle=
    ”http://schemas.xmlsoap.org/soap/encoding/>
  <SOAP-ENV:Body>
     <m:GetLastTradePriceRequest xmlns:m=”Some-URI”>
        <symbol>MOT</symbol>
     </m: GetLastTradePriceRequest>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP and XML (response)

```
<HTTP/1.1 200 OK
Content-Type: text/xml; charset=”utf-8”
Content-Length: nnnn

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope/”
    SOAP-ENV:
      encodingStyle=”http://schemas.xmlsoap.org/soap/encoding/”>
    <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse  xmlns:m=”Some-URI”>
         <price>34.5</price>
       </m: GetLastTradePriceResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP and XML (error)

```
<HTTP/1.1 200 OK
Content-Type: text/xml; charset=”utf-8”
Content-Length: nnnn
<SOAP:Envelope
XMLns:SOAP="HTTP://schemas.XMLSOAP.org/SOAP/envelope"
SOAP:encodingStyle=
                    "HTTP://schemas.XMLSOAP.org/SOAP/encoding">
    <SOAP:Body>
        <SOAP:Fault>
            <faultcode>Client</faultcode>
            <faultstring>Invalid Request</faultstring>
            <faultactor>unknown</faultactor>
            <detail>Requested stock…</detail>
        </SOAP:Fault>
    </SOAP:Body>
</SOAP:Envelope>
```

Web Services

First definition:

WS (Web Services)

Platform and **implementation independent software components** that can be:

- **described** using a service description language (**WSDL**)
- invoked using a **remote API**, usually over the network (**SOAP**)
- (**published in a service registry** (**UDDI**))

  we will not present this aspect `rmiregstry`?!?

Web Services: WSDL

For WS, in addition to **communication**…

We need a mechanism to describe both **abstract and concrete service aspects**

WSDL (Web Services Description Language)

A XML proposal to **describe Web Services and publishing them** specifying the **message format for** both **requests** and **responses** in a **standard and portable** way.

WDSL specifies:

- **what a service can do** (requests, responses and parameters)
- **where it resides**
- **how to invoke it**

Web Services Description Language

To use an unknown Web service:

– get the **WSDL file**

– analyze the **WSDL document** to obtain

- **service location**
- **method names and parameters**
- **how to access methods**

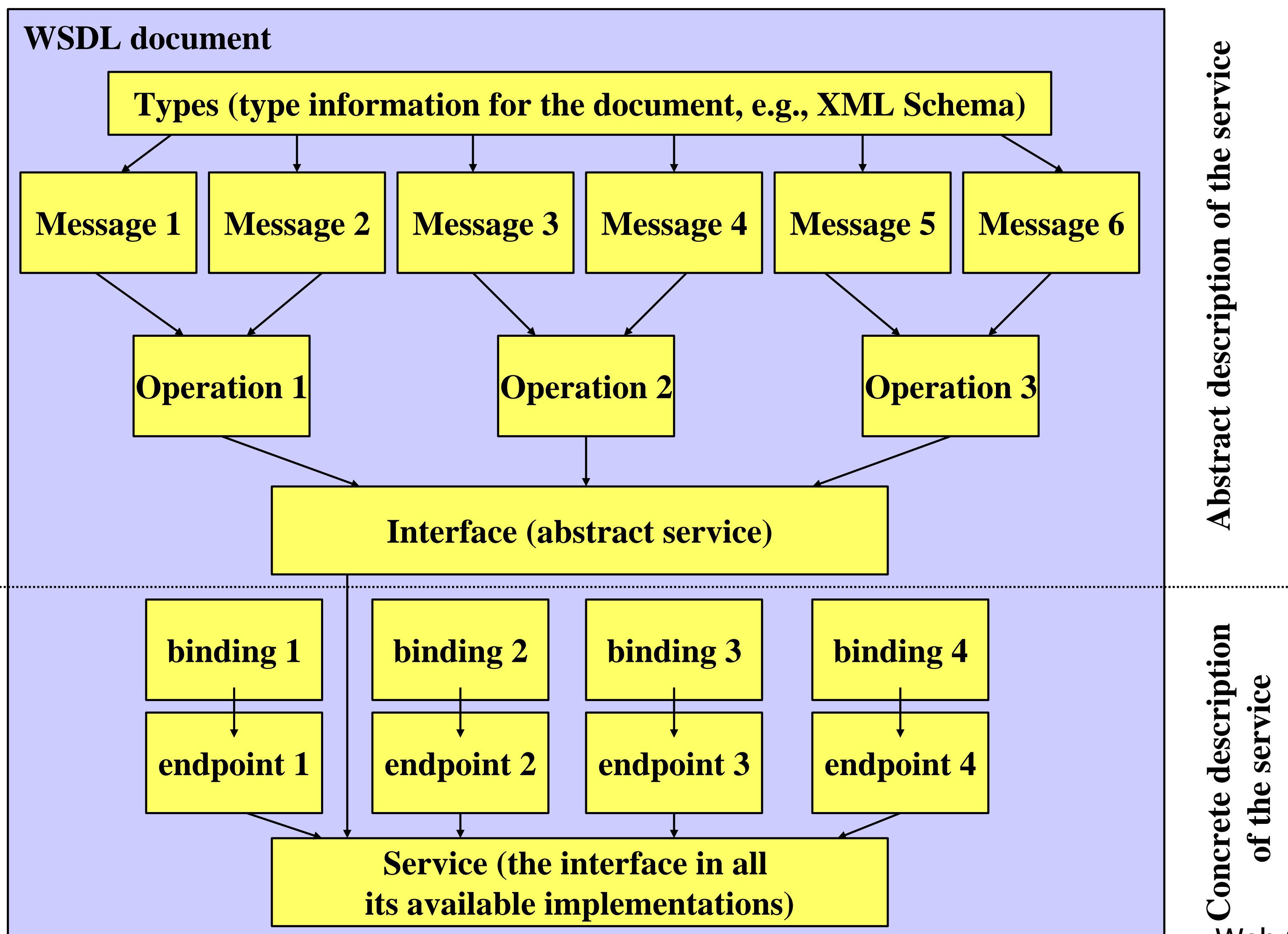
– build a **SOAP request**

– send the **SOAP request to the service** and wait for a **response**

The rationale is to have a broad support and many facilities, up to the complete automation by a middleware

Some parts of WSDL are similar to an IDL

WSDL basic elements



WSDL v 2.0 Architecture

WSDL describes Web Services starting from the message exchange between Requestor and Provider

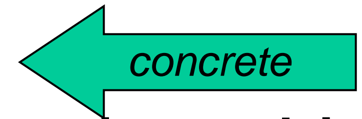
*Messages are described first in an **abstract** form, then in a more **concrete** way (protocol and format)*

A message is a collection of typed elements

An operation is a message exchange

An interface (portType v.1) is a collection of operations

A service is the implementation of an interface and it contains an endpoint collection (port v. 1)

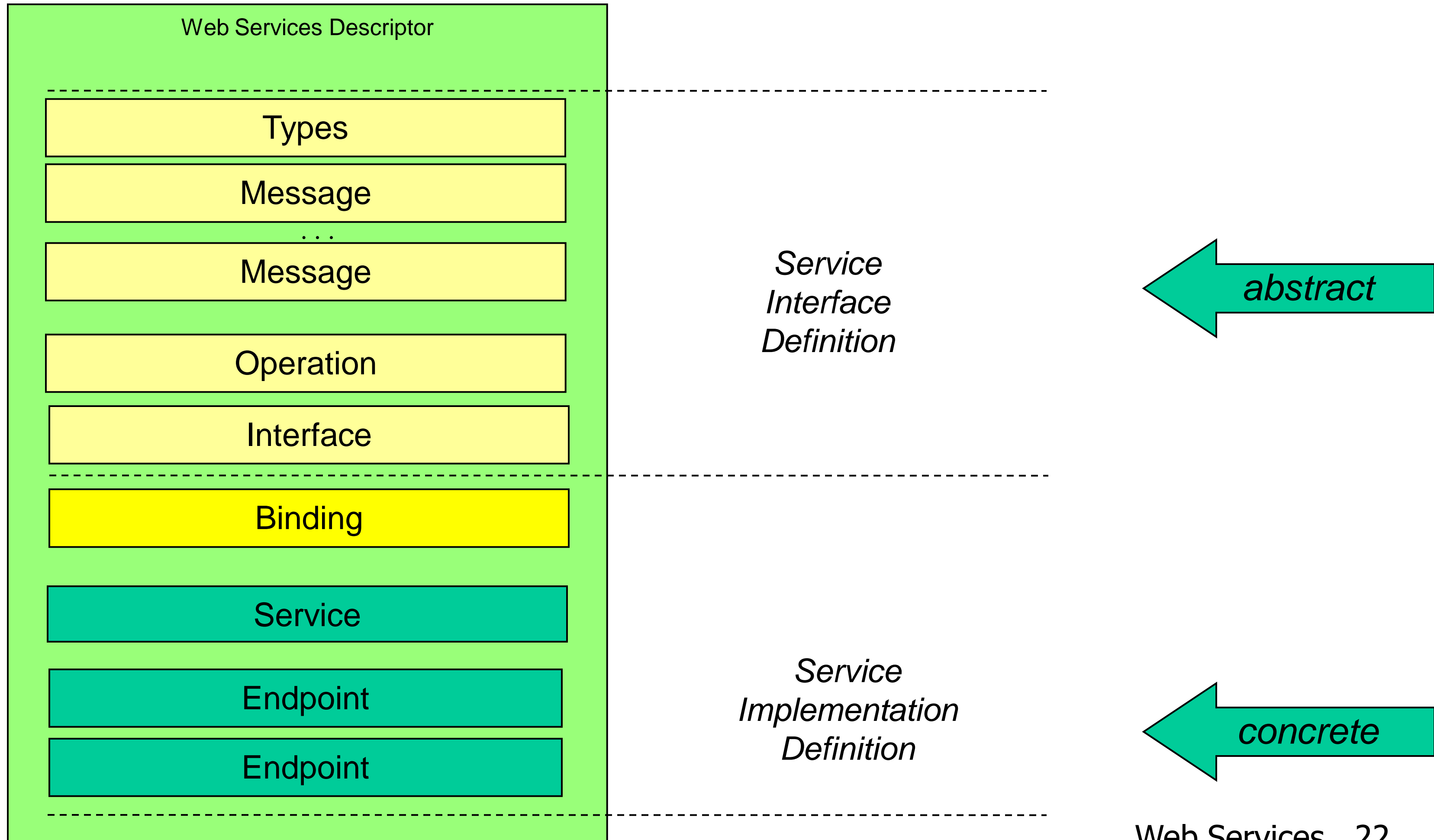


An endpoint is the concrete implementation of the service and it includes all the concrete details needed for successful communication

A binding is the link needed to request concrete operations

WSDL 2.0

WSDL describes abstract and concrete aspects



A SERVICE in WSDL

A WSDL document is composed by:

– **Abstract** parts

Type, **Message**, **Operation**, **Interface**

– **Concrete** parts

Binding, Endpoint, Service

WSDL defines first abstract elements, then the relative concrete elements

The **abstract** version of the service is generalized, flexible and easily extensible.

The **concrete** details are specified in **each element that take part in the service**

Abstract elements in WSDL

- **type**

A data type in a message using XML Schema

- **message**

Information actually exchanged between requestor and provider, specialized as input, output, and fault messages

- **operation**

Specification of the **name** of a operation, its **input** and **output parameters** and is composed by **messages**

- **interface**

A set of **abstract operations** and **messages**, identified by a unique id, that corresponds to the service itself, and is unique in a WSDL document.

Concrete parts in WSDL

- **binding** details of the **implementation** and **operations** contained in an interface

Specifies the actual protocol: transport and data coding

(**HTTP**, **SOAP**; **SMTP**; **FTP**; …)

- **endpoint** identifies the network **address** of the service

- **service** **collection** of related endpoints

It allows to group interfaces to emphasize the endpoints supported by a service.

For example, *all the endpoints associated to a transaction that requires multiple steps*

Web Services: WSDL types

The first section of a WSDL document describes the abstract data types needed by operations

```
<types> <schema>
    <element name="TradePriceRequest">
      <complexType>
        <all>
         <element name="symbol" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePriceResponse">
       <complexType>
         <all>
           <element name="price" type="float"/>
         </all>
       </complexType>
      </element>
</schema> </types>
```

WSDL message, operation, and interface

Then, the **messages and operations** description:

```
<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
 </message>
<message name="GetLastTradePriceOutput">
 <part name="body" element="xsd1:TradePrice"/>
</message>
```

Each operation comprises a request and a response message, grouped into an interface

```
<interface name="StockQuoteInterface">
  <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</interface>
```

WSDL binding

A binding is a link between the interface name (type), one or more operation names (name) and actions to execute (soapAction):

```
<binding name="StockQuoteSoapBinding"
 type="tns:StockQuoteInterface">
  <soap:binding>
    <operation name="GetLastTradePrice">
      <soap:operation

soapAction="http://lia.deis.unibo.it/soap/bin/"/>
      <input><soap:body use="literal"/><input>
      <output><soap:body use="literal"/></output>
    </operation>

   …
</binding>
```

They refer to concrete implementation

WSDL endpoint and service

The last part of a WSDL document describes the service and the Web server to use to access it:

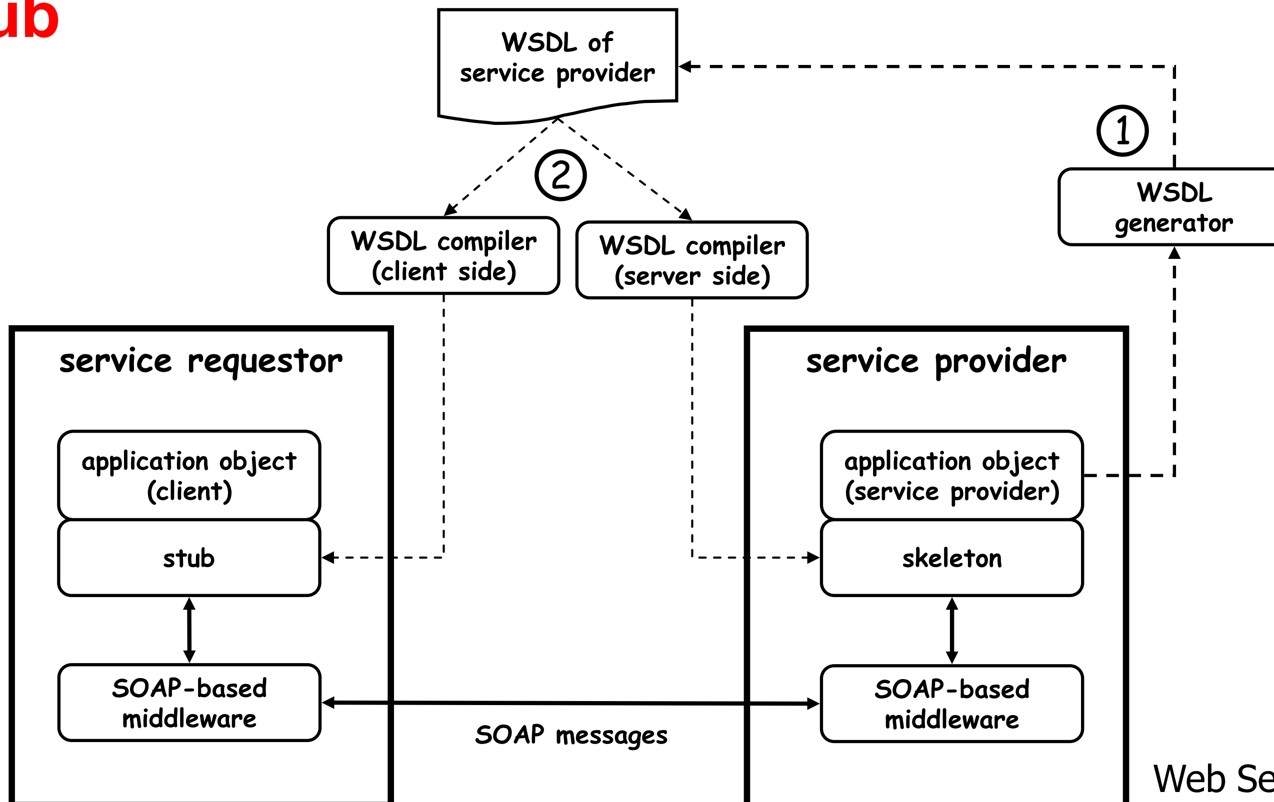
```
<service name="StockQuoteService">
  <documentation>
    Stock exchange service
  </documentation>
   <endpoint name="StockQuoteEndPoint"
     binding="tns:StockQuoteBinding">
   <soap:address location="http://www.stockquote.com"/>
  </endpoint>
</service>
```

In addition it describes all needed concrete details

WSDL usage

WSDL may be used as:

- **description of the service contract IDL**
- **IDL specification → starting point to compile the (Client-side) stub**



Development process

Environment setup

1. Download the file *labWS.zip*, extract it, and set the destination directory as current directory (***labWS***)
2. Open the file setupEnv.bat and adapt it to your working environment by setting the appropriate variables
   - LAB_HOME=<full path to labWS directory>
   - JAVA_HOME=<full path to JDK directory>
3. Run (in the current directory) setupEnv.bat from the command prompt (this step completes the environment setup)

EchoInterface service: Java interface

/* We already know Java RMI, let's see the service declaration as Java RMI remote interface */

```
public interface EchoInterface extends java.rmi.Remote {
    public String echoString(String in0)
                         throws RemoteException;
    public String appendString(String in0, String in1)
                         throws RemoteException;
    public int sum(int in0, int in1)
                         throws RemoteException;
}
```

… there are tools (e.g. AXIS Java2WSDL) that automatically generate a WSDL document from a Java interface

EchoInterface WSDL: WSDL Types

// Let us see now, for the same interface the WSDL document

```
<wsdl:message name="appendStringRequest">
  <wsdl:part name="in0" type="soapenc:string"/>
  <wsdl:part name="in1" type="soapenc:string"/>
</wsdl:message>
</wsdl:message>
<wsdl:message name="appendStringResponse">
  <wsdl:part name="appendStringReturn" type="soapenc:string"/>
</wsdl:message>
<wsdl:message name="echoStringResponse">
  <wsdl:part name="echoStringReturn" type="soapenc:string"/>
</wsdl:message>
<wsdl:message name="echoStringRequest">
  <wsdl:part name="in0" type="soapenc:string"/>
<wsdl:message name="sumRequest">
  <wsdl:part name="in0" type="xsd:int"/>
  <wsdl:part name="in1" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="sumResponse">
  <wsdl:part name="sumReturn" type="xsd:int"/>
</wsdl:message>
```

EchoInterface WSDL: interface (WSDL v. 1 portType)

```
<wsdl:portType name="EchoInterface">
 <wsdl:operation name="echoString" parameterOrder="in0">
  <wsdl:input message="impl:echoStringRequest"
              name="echoStringRequest"/>
  <wsdl:output message="impl:echoStringResponse"
               name="echoStringResponse"/>
 </wsdl:operation>
 <wsdl:operation name="appendString" parameterOrder="in0 in1">
  <wsdl:input message="impl:appendStringRequest"
              name="appendStringRequest"/>
  <wsdl:output message="impl:appendStringResponse"
               name="appendStringResponse"/>
  </wsdl:operation>
 <wsdl:operation name "sum" parameterOrder="in0 in1">
  <wsdl:input message="impl:sumRequest" name="sumRequest"/>
  <wsdl:output message="impl:sumResponse" name="sumResponse"/>
 </wsdl:operation>
</wsdl:portType>
```

EchoInterface WSDL: binding

```
<wsdl:binding name="EchoServiceSoapBinding"
              type="impl:EchoInterface">
  <wsdlsoap:binding style="rpc
          transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="echoString">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="echoStringRequest">
      <wsdlsoap:body
         encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
         namespace="urn:EsempioEchoService" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="echoStringResponse">
      <wsdlsoap:body
         encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
         namespace="urn:EsempioEchoService" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
...
</wsdl:binding>
```

EchoInterface WSDL: service and endpoint (WSDL v. 1 port)

```
<wsdl:service name="EchoInterfaceService">
  <wsdl:port binding="impl:EchoServiceSoapBinding"
             name="EchoService">
    <wsdlsoap:address location=
     "http://192.168.1.100:8079/axis2/services/EchoService"/>
  </wsdl:port>
</wsdl:service>
```

EchoInterface WS Java invocation

We will use Apache AXIS library

It offers a set of tools for Web service development both client and server side

In this lab, we will focus on the client side only

We will see two different implementation strategies:

- Direct call construction
- Automatic code generation (WSDL2Java compiler) and stub usage for the remote invocation → *similar to RMI*

Abstractions and Tools provided by Apache AXIS

AXIS library provides several abstractions to simplify the development process of WSs in Java

- ***Service:*** **a generic WS**
- ***Call:*** **a single invocation (RPC like) of a remote operation**
- ***QName:*** **an "*XML qualified name*" composed by an URL that identifies the reference XML namespace and a local name within the namespace**

In addition… tools for automatic client stub generation: WSDL2Java

ClientBuildCall.java 1/2

```
import javax.xml.namespace.QName;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

public class ClientBuildCall {
 public static void main(String[] args) {
  try
  { String endpoint =   // Set the remote endpoint as full URL
        "http://192.168.1.100:8079/axis/services/EchoService";
    Service service = new Service();
    Call call = (Call) service.createCall(); // Build call
    // Initialize the call object specifying the target endpoint
    call.setTargetEndpointAddress(new java.net.URL(endpoint));
    call.setOperationName(
      // QName is the XML qualified name that references the requested operation
      // specified within the WSDL document
      new QName("http://192.168.1.100:8079/axis/services/EchoService",
                "echoString") );
```

ClientBuildCall.java 2/2

```
      /* For multiple input/output parameters, AXIS uses java Object arrays
       * Note: AXIS library automatically executes several support actions
       * 1) converts the input from the locale format (Java) to serialized XML text;
       * 2) wraps and sends the SOAP request;
       * 3) receives and extracts the SOAP response;
       * 4) converts the output from serialized XML text to local format (Java);
       */
      String ret = (String) call.invoke(new Object[]{"Hello!"} );
      System.out.println("Result: " + ret);
    } catch (Exception e) { System.err.println(e.toString()); }
  }    // main
}    // ClientBuildCall
```

Compilation and Execution step-by-step

1. Change current directory to the directory containing source code:

```
> cd src
```

2. Compile:

```
> javac ClientBuildCall.java
```

3. Execute client:

```
> java ClientBuildCall
```

Try to call other services, by changing the source code and repeating steps 2 and 3

ClientStubCall.java

```
import java.net.URL;
import org.apache.axis.client.Service;
import EchoExample.EchoServiceSoapBindingStub;

public class ClientStubCall {
 public static void main(String[] args) {
  try  // Use client stub automatically generated by WSDL2Java compiler
  { EchoServiceSoapBindingStub service =
     new EchoServiceSoapBindingStub(
     // We only need to identify the endpoint as URL
     new URL("http://137.204.45.59:8079/axis/services/EchoService"),
               new Service());
    // The stub provides remote operations as local methods → similar to RMI Stub
    String result = service.echoString("Hello!");
    System.out.println("Result: " + result);
  } catch (Exception e) {
    System.err.println(e.toString());
  }

 }   // main
} // ClientStubCall
```

Compilation and Execution step-by-step

1. Change current directory to the directory containing source code:

```
> cd src
```

2. Compile the stub using the script generateClasses.bat

```
> generateClasses.bat EchoService.wsdl
```

3. Compile client:

```
> javac ClientStubCall.java
```

4. Execute client:

```
> java ClientStubCall
```

Try to call other services, by changing the source code and repeating steps 3 and 4