Distributed computing systems

Web Services

Web Services API Styles

Web-service style	Problem
RPC API	How clients can execute remote procedures via HTTP?
Resource API (REST)	How a client can manipulate data provided by the remote system over HTTP, without binding itself to remote procedures, and without having to create a special problem-oriented API?

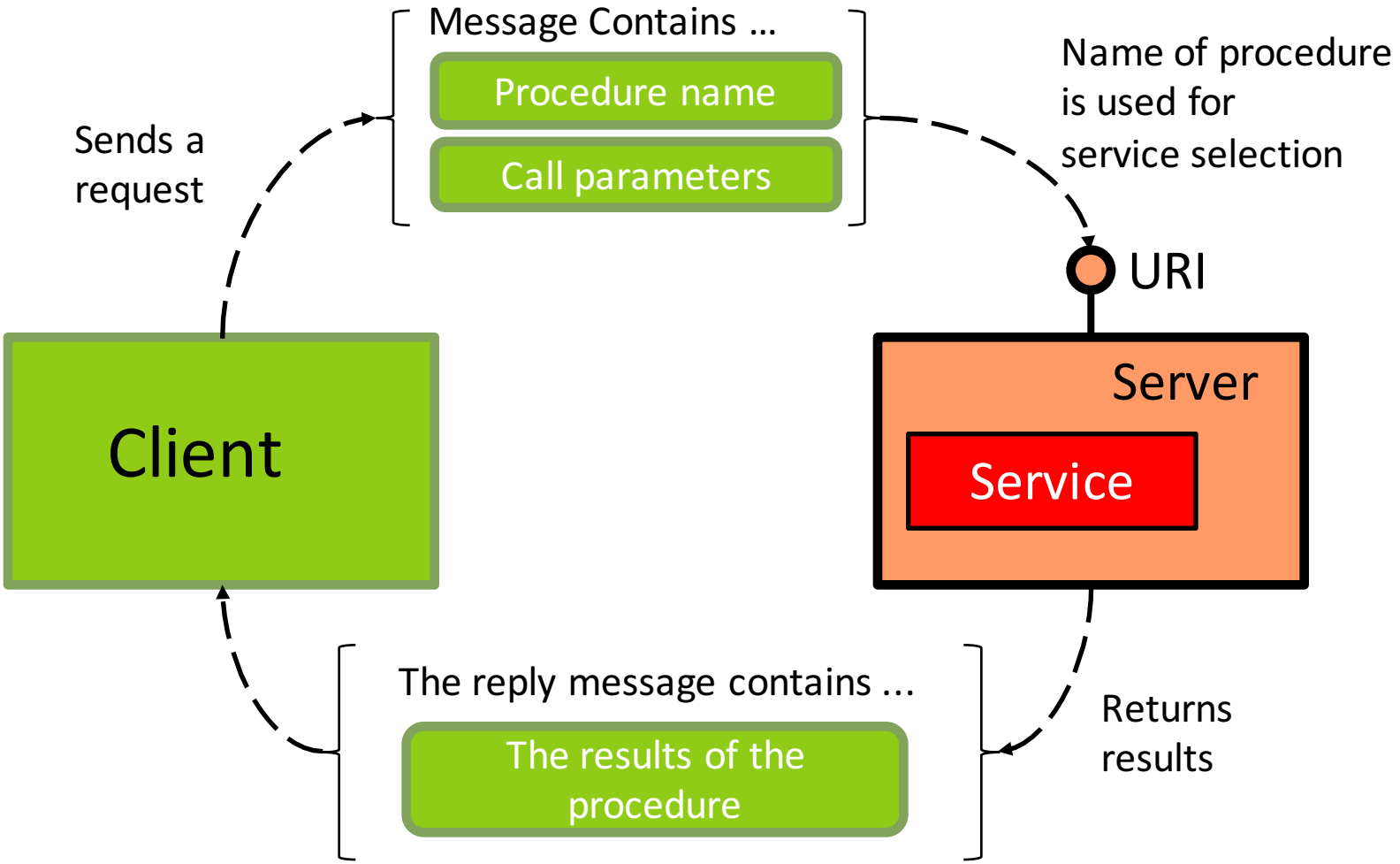
RPC API

Gleb Radchenko, SUSU

RPC API

- You identify the messages that describe:
  - *remote procedures,*
  - and *a set of parameters* for the remote procedure.
- The client must send a message with this information to the specified URL to perform the procedure.

RPC API


Message Contains …
Procedure name
Call parameters
Name of procedure is used for service selection
Sends a request
URI
Server
Service
Client
The reply message contains …
The results of the procedure
Returns results

RPC PLATFORMS

- Use of HTTP protocol solves many of “Classic RPC” (pre-web) problems, because allows the client and server interaction, based on different platforms via the Internet.
- There are many implementations for organization of RPC over HTTP in different languages:
  - JAX-WS (Java)
  - WCF (Microsoft .NET)
- They provide means for delivering of simple remote procedures and Web clients to them without having to understand the data format (XML, JSON), encoding methods (UTF-8, etc.) or service description structure (WSDL).

RPC Implementation

- There are two main approaches for describing such services contracts:
  - *XML Web Services:* the most common approach uses the WSDL (Web Services Description Language) and WS-Policy specification and WS-Security to define different authentication methods, encryption and others.
  - *Non-XML Web Services (RPC):* it is a whole range of different RPC protocols, starting with the JSON-RPC standard, and a variety of finishing proprietary or open standards from different suppliers.

JSON-RPC

- JSON-RPC (JavaScript Object Notation Remote Procedure Call) — is a Remote Procedure Call protocol that uses JSON for message encoding.
- JSON-RPC works by sending requests to the server that implements the protocol. Client is usually the program, which you want to invoke a method on a remote system. All transmitted data - simple objects serialized to JSON

JSON-RPC – REQUEST AND RESPONSE

- **The request** must contain three essential properties:
  - *method* — A string with the name of the called method.
  - *params* — An array of objects to be transmitted to the method as parameters.
  - *id* — The value of any type, which is used to set the correspondence between the request and response.

- **The response** should contain the following properties:
  - *result* — The data that is returned by a method method. If an error occurs during the execution of the method, this property must be set to null.
  - *error* — The error code if an error occurs during the execution of a method, or null.
  - *id* — Same value as in the request.

JSON-RPC – EXAMPLE

--> means the data sent to the server (request)

<-- represents a response

```
...
--> {"method": "postMessage", "params": ["Hello all!"], "id": 99}
<-- {"result": 1, "error": null, "id": 99}
<-- {"method": "handleMessage", "params": ["user1", "Ok!"], "id": null}
<-- {"method": "handleMessage", "params": ["user3", "gotta go"], "id": null}
--> {"method": "postMessage", "params": ["I have a question:"], "id": 101}
<-- {"method": "userLeft", "params": ["user3"], "id": null}
<-- {"result": 1, "error": null, "id": 101}
...
```

JSON-RPC Implementation

More than 100 libraries for all the most common languages

Name	JSON-RPC version	Description	Language(s)
Jean-Rpc UNIX Router	2.0	A UNIX IRouter implementation for Json Rpc v2 requests for Microsoft.AspNet.Routing (frameworks: dnx451, dnxcore50).	.NET
JSON-RPC.NET	2.0	A fast, open-source JSON-RPC 2.0 server. Supports sockets, pipes, and HTTP with ASP.NET. Requires Mono or .NET Framework 4.0.	.NET
Jayrock	1.0	A server implementation of JSON-RPC 1.0 for versions 1.1 and 2.0 of Microsoft's .NET Framework.	.NET
jsonrpc-c	2.0	C library for JSON-RPC on TCP sockets (server only)	C
jsonrpc	2.0	Transport-independent JSON-RPC server with parameters validation via jansson	C
libjrpc	2.0	Lightweight JSON-RPC 2.0 client and server library	C
libjson-rpc-cpp	1.0+2.0	Open source JSON-RPC framework for C++, including client/server support via HTTP and a stub generator for C++ and JavaScript	C++
Json-Rpc-Cpp	2.0	Open source JSON-RPC implementation in C++	C++
gSOAP	2.0	Open source JSON/JSON-RPC implementation in C/C++, includes code generators for XML/SOAP and JSON/JSON-Path	C/C++
Proboss	2.0	Implementation for Qt/C++. Abstracts the communication layer (there are TCP and HTTP classes ready to use, also).	C++
qjsonrpc	2.0	Implementation for Qt/C++. Supports connection between the messages and Qt's signal/slot system (QObjs, qdrpc) and utilizes the new JSON classes included as part of Qt5.	C++
[illegible]	[illegible]	[illegible]	C++
Any-RPC	[illegible]	[illegible]	C++
jsonrpc-lean	2.0	Lightweight, fast, transport-agnostic, c++11 implementation of the JSON-RPC 2.0 specification	C++
cljl-json-rpc	2.0	Full duplex async JSON-RPC 2.0 library upon xstream transport (such as TCP and TCP with TLS).	Clojure
jsonrpc2-dart	2.0	Implementation for Dart. Server methods are transport-agnostic. Client is HTTP only.	Dart
json_rpc_2	2.0	Implementation for Dart. [illegible]	Dart
Superobject (was JSON Toolkit)	[illegible]	[illegible]	Delphi
jsonrpc2-erlang	2.0	A minimalistic Erlang implementation that supports concurrent batch requests. Complete, but does nothing besides JSON-RPC 2.0. In particular, JSON encoding and decoding must be performed by the user.	Erlang
go/rsel/rpc	?	Standard Go library JSON-RPC implementation	Go
Gorilla web toolkit	1.0+2.0	Gorilla is a web toolkit for the Go programming language.	Go
json-rpc-server	2.0	An implementation of the server side of JSON RPC 2.0	Haskell
hs-json-rpc	1.0+2.0	A library for writing JSON-RPC client applications in Haskell	Haskell
Brutusin-RPC	2.0	Released on 2016. JSON-RPC 2.0 over HTTP and Websockets. Built for maintainability. Easily exposes Java actions as self describing, documented services. Includes builtin service repository and functional testing module	Java
com-pale	2.0	JSON-RPC 2.0/HTTP, REST/HTTP supporting framework that runs on web application servers. POJO, Spring, EJB like objects can be easily exposed.	Java
jsonrpc4j	2.0	Java implementation JSON-RPC 2.0 supporting streaming as well as HTTP servers. It also has support for spring service exporter/consumer.	Java
json-rpc	1.0	Generic Java/JavaScript implementation which integrates well on Android/Servlets/Standalone Java/JavaScript/App-Engine applications.	Java / JavaScript
JSON Service	2.0	JSON-RPC protocol implementation (server-side) in Java with Service Mapping Description support. It integrates well with Dojo Toolkit and Spring Framework.	Java
JSON-RPC 2.0	2.0	A minimalist Java library for parsing, representing and serializing JSON-RPC 2.0 messages (open source). Multiple implementations on the site. (Base, Client, Shell, ...)	Java
java-json-rpc	2.0	Implementation for J2EE servers.	Java
lib-json-rpc	2.0	Implementation on servlet, client, JavaScript	Java
simple-jsonrpc	2.0	Another simple JSON-RPC 2.0 servlet, servicing the methods of a class.	Java
json-rmi	2.0	Light-weight, transport-independent, extensible RMI framework geared towards distributed computing	Java
JDBCWizard	2.0	Generates JSON-RPC 2.0 services for calling PL/SQL and SQL statements in Oracle databases	Java
jsonserv	2.0	A lightweight, JAR-packaged, self-describing, JSON-RPC service framework for J2E (servlet-based) for easily exposing business methods through a JSON over HTTP API. Aimed at creating AJAX/JSON web interfaces	Java
jrpc2	2.0	Full featured, modular JSON-RPC 2.0 library with support of batches and named parameters. Server&Client. Features: Express, Koa, Socket.IO middlewares + HTTP, TCP, ZeroMQ transports.	JavaScript, Node.js, io.js
jsonrpcjs	1.0(2.0)	JavaScript client library for JSON-RPC 1.0, supports call batching has no dependency on external libraries. Main version does not contain support for named parameters, but on the GitHub is pull request version to support JSON-RPC 2.0 (only) [1]	JavaScript
Raptor RPC	2.0	A transport-agnostic RPC server with middleware support and an easy-to-use API.	JavaScript
easyXDM	2.0	Library for cross-domain messaging with a built-in RPC feature. The library supports all web browsers by using a mix of postMessage, nix, frameElement, window.name, and FIM, and is very easy to use.	JavaScript
Dojo Toolkit	1.0+	Offers a broad support for JSON-RPC	JavaScript
Pmrpc	2.0	An inter-window and Web Worker remote procedure call JavaScript library for use within HTML5 browsers. Pmrpc is an implementation of JSON-RPC using the HTML5 postMessage API for message transport.	JavaScript
goxoa	2.0	Includes a JSON-RPC implementation with optional backends in Java, PHP, Perl and Python.	JavaScript, Java, PHP, Perl, and Python
JSON-RPC implementation in JavaScript	2.0	Includes JSON-RPC over HTTP and over TCP/IP sockets	JavaScript
jabsorb	2.0	A lightweight Ajax/Web 2.0 JSON-RPC Java framework that extends the JSON-RPC protocol with additional ORB functionality such as circular references support.	JavaScript, Java
The Wakanda platform	2.0	Includes a JSON-RPC 2.0 client in its Ajax Framework and a JSON-RPC 2.0 service in server-side JavaScript	JavaScript
Delmos	1.0+2.0	Server implementation for Node.js/JavaScript.	JavaScript
jQuery JsonRpcClient	2.0	JSON-RPC 2.0 client for HTTP and WebSocket backends	JavaScript
AF JSONRPCClient	2.0	JSON RPC Client addition to [AF Networking] 2.0	Objective-C
DeferredKit	1.0	Includes a JSON-RPC 1.0 client.	Objective-C
Demiurgic	2.0	JSON-RPC 2.0 client for Objective-C	Objective-C
Oxen iPhone Commons JSON component	1.0	JSON-RPC 1.0 client for Objective-C	Objective-C
objc-JSONRpc	2.0	An Objective-C JSON-RPC client. Supports notifications, single calls and multicalls	Objective-C
JSON::RPC	2.0	JSON-RPC 2.0 server implementation	Perl
json-rpc-perl	2.0	Client and server with dispatch to multi methods, support of positional/named params, notifications, batches and extensible error handling.	Perl 6
CSJRPC	2.0	Simple PHP (Server only) implementation of JSON-RPC 2.0. Based on jsonrpc	PHP
php-json-rpc	2.0	Simple PHP implementation of a JSON-RPC 2.0 over HTTP client.	PHP

XML Web services

XML Web services

- XML Web Services – is XML-based platform-independent technology that supports distributed computing.
- XML Web services are software components that allow you to create independent, scalable, loosely coupled applications.
- Their operation is based on the use of HTTP, XML, XSD, SOAP, and WSDL protocols.
- SOAP/ WSDL frameworks:
  - The Java API for XML Web Services (JAX-WS)
  - Apache CXF
  - Microsoft's Windows Communication Foundation (WCF)
  - …

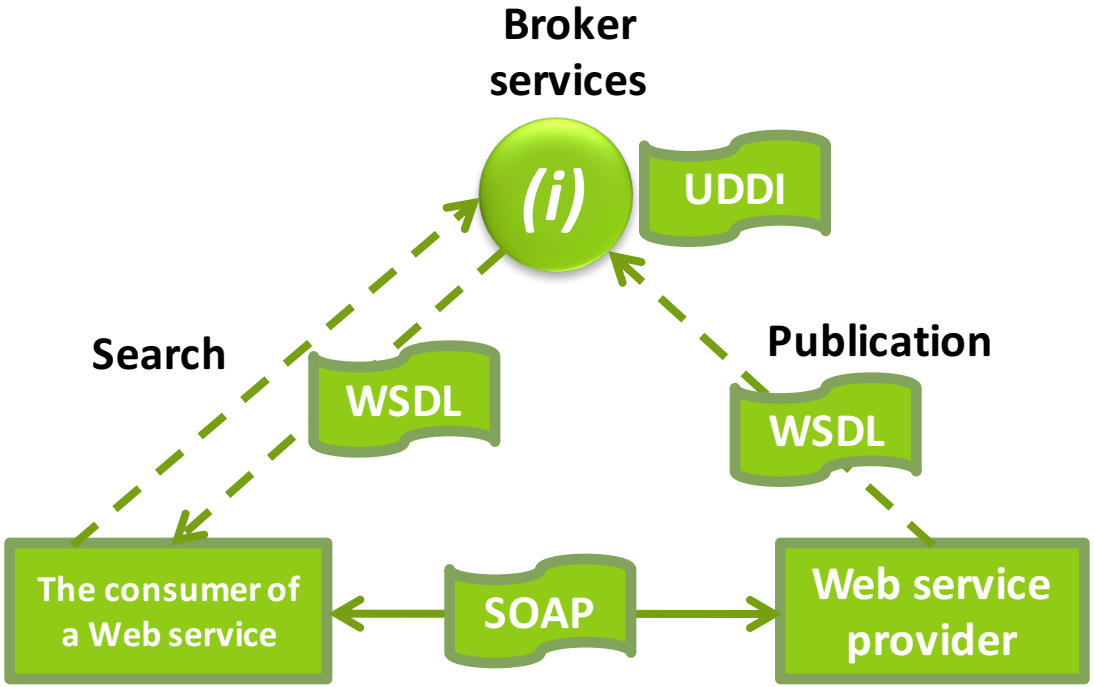
The WSDL: Web Service Definition Language

- WSDL is a standard XML document describing fundamental properties of the Web service, such as:
  - **What is it** - a description of the methods provided by the Web service;
  - **How do I access** - data format and protocols;
  - **Where it is located** - a special network address of the service.

SOAP (not only Simple Object Access Protocol)

- SOAP - a protocol based on the exchange of XML-documents.
- SOAP is defined as follows: "SOAP is an XML-based protocol for exchanging information in a decentralized, distributed environment.

"Web services Triangle"



Addressing Web services

- Web service address is a standard URI (Uniform Resource Identifer).

```
http://live.capescience.com/ccx/GlobalWeather
```

  - BTW: URL (Uniform Resource Locator) is a type of URI.

Software environment for Web services development

- Apache Axis, on the basis of, for example, Apache Tomcat
- IBM Websphere
- Microsoft .NET
- J2EE (Java 2 Enterprise Edition) server container

The WSDL Standart

Elements of the WSDL document

- *Service* - a named collection of (endpoints) ports
- *Port* - the address or connection point to a Web service.
- *PortType* - methods provided by the Web service
- *Binding* - communication protocols used by the Web service
- *Types* - data types used by the Web service
- *Message (n/a)* - messages used by the Web service.

Example of a Web service

- A simple example of a Web service (Java)

```
@WebService()
 public class AcmeCalculator {
    @WebMethod
    public int Summ(int a, int b) {
        int result = a + b;
        System.out.println(result);
        return result;
    }
    public static void main(String[] argv) {
        Object implementor = new AcmeCalculator();
        String address = "http://localhost:9000/AcmeCalculator";
        Endpoint.publish(address, implementor);
    }
}
```

AcmeCalculator.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions     targetNamespace="http://example" name="AcmeCalculator"
                 xmlns="http://schemas.xmlsoap.org/wsdl/"
                 xmlns:wsp="http://www.w3.org/ns/ws-policy"
                 xmlns:tns="http://example"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
                 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                 xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
                 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd>

  <import namespace="http://example/" location="AcmeCalculatorPortType.wsdl"/>

  <binding name="AcmeCalculatorBinding" type="ns1:AcmeCalculator" xmlns:ns1="http://example/">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="Summ">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="AcmeCalculator">
    <port name="AcmeCalculator" binding="tns:AcmeCalculatorBinding">
      <soap:address location="http://localhost:8080/services/example/AcmeCalculator"/>
    </port>
  </service>

</definitions>
```

Namespaces

Binding

Service definition

AcmeCalculatorPortType.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions    targetNamespace="http://example/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://example/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<types>
    <xsd:schema>
        <xsd:import namespace="http://example/" schemaLocation="AcmeCalculatorPortType_schema1.xsd"/>
    </xsd:schema>
</types>
<message name="Summ">
    <part name="parameters" element="tns:Summ"/>
</message>
<message name="SummResponse">
    <part name="parameters" element="tns:SummResponse"/>
</message>
<portType name="AcmeCalculator">
    <operation name="Summ">
        <input  ns1:Action="http://example/AcmeCalculator/SummRequest"
                message="tns:Summ“
                xmlns:ns1="http://www.w3.org/2007/05/addressing/metadata"/>

        <output ns2:Action="http://example/AcmeCalculator/SummResponse"
                message="tns:SummResponse"
                xmlns:ns2="http://www.w3.org/2007/05/addressing/metadata"/>
    </operation>
</portType>
</definitions>
```

AcmeCalculatorPortType_schema1.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<xs:schema   version="1.0"
         targetNamespace="http://example/"
         xmlns:tns="http://example/"
         xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="Summ" type="tns:Summ"/>
<xs:element name="SummResponse" type="tns:SummResponse"/>

<xs:complexType name="Summ">
   <xs:sequence>
      <xs:element name="arg0" type="xs:int"/>
      <xs:element name="arg1" type="xs:int"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="SummResponse">
   <xs:sequence>
      <xs:element name="return" type="xs:int"/>
   </xs:sequence>
</xs:complexType>

</xs:schema>
```

WSDL Ports <portType>

- Element <portType> is the most important element in WSDL.
- It defines a Web service itself, its operation, and messages.
- Can be compared to a library of functions, where all the input parameters and the results of the functions are defined.

An example of a <portType> block

```
<portType name="AcmeCalculator">
   <operation name="Summ">
      <input ns1:Action="http://example/AcmeCalculator/SummRequest"
           message="tns:Summ"
           xmlns:ns1="http://www.w3.org/2007/05/addressing/metadata"/>

      <output ns2:Action="http://example/AcmeCalculator/SummResponse"
           message="tns:SummResponse"
           xmlns:ns2="http://www.w3.org/2007/05/addressing/metadata"/>
   </operation>
</portType>
```

WSDL Messages <message>

- The <message> element defines the data elements of the operation.

- Each message can contain one or more parts. These parts can be compared to the parameters of the called functions in traditional programming languages.

An example of a <message> block

```
<message name="Summ">
  <part name="parameters" element="tns:Summ"/>
</message>

<message name="SummResponse">
  <part name="parameters" element="tns:SummResponse"/>
</message>
```

The WSDL bindings <binding>

- The <binding> element defines the message format and protocol details for each port.

- Responsible for the manner in which the elements of an abstract interface in the <portType > block converted into arrays of information in the format of interaction protocols such as SOAP.

An example of a <binding> block

```
<binding name="AcmeCalculatorBinding" type="ns1:AcmeCalculator"
xmlns:ns1="http://example/">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
    <operation name="Summ">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
```

<PORT> AND <SERVICE> BLOCKS

- These blocks determine where the service is located.
- port describes the location and access to the endpoint
- service is a named collection of ports

An example of a <service> block

```
<service name="AcmeCalculator">
    <port name="AcmeCalculator" binding="tns:AcmeCalculatorBinding">
      <soap:address
location="http://localhost:8080/services/example/AcmeCalculator"/>
    </port>
</service>
```

WSDL-based Client generation

```
@WebService(name = "AcmeCalculator", targetNamespace = "http://example/")
@XmlSeeAlso({
    ObjectFactory.class
})

public interface AcmeCalculator {
    /**
     *
     * @param arg1
     * @param arg0
     * @return
     *     returns int
     */
    @WebMethod(operationName = "Summ")
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "Summ", targetNamespace = "http://example/", className =
"AcmeCalculatorClient.Summ")
    @ResponseWrapper(localName = "SummResponse", targetNamespace = "http://example/", className
= "AcmeCalculatorClient.SummResponse")
    @Action(input = "http://example/AcmeCalculator/SummRequest", output =
"http://example/AcmeCalculator/SummResponse")
    public int summ(
        @WebParam(name = "arg0", targetNamespace = "")
        int arg0,
        @WebParam(name = "arg1", targetNamespace = "")
        int arg1);

}
```

Client invocation

```
public class AcmeCalculatorClient {
    public static void main(String[] argv) {
        AcmeCalculatorService calculatorService = new AcmeCalculatorService();
        AcmeCalculator calculator = calculatorService.getAcmeCalculatorPort();
        int result = calculator.summ(5,6);
        System.out.println(result);
    }
}
```

The SOAP Standard

SOAP Message

- A SOAP message is a one-way transfer of information between SOAP nodes: from the source to the receiver.
- SOAP messages are a fundamental building blocks for more complex interaction patterns:
  - Request/response
  - "dialog" mode
  - etc.

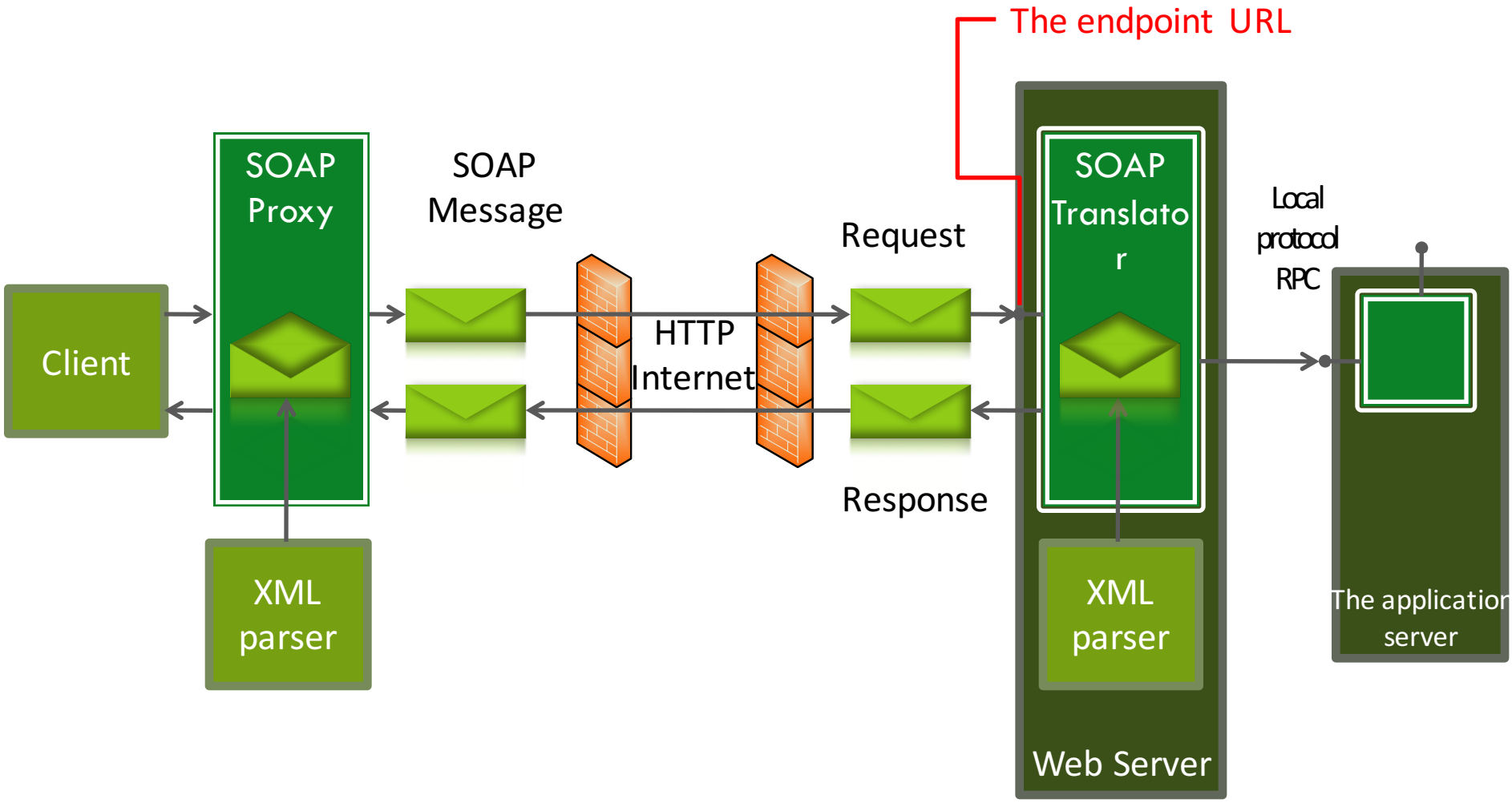
The SOAP Structure

- SOAP consists of 3 parts:
  - wrapper that defines the environment to describe the content of the message and how to handle it;
  - a set of encoding rules for expressing the essence of the data types defined in the application;
  - Convention of remote procedures call and responses obtaining.

SOAP

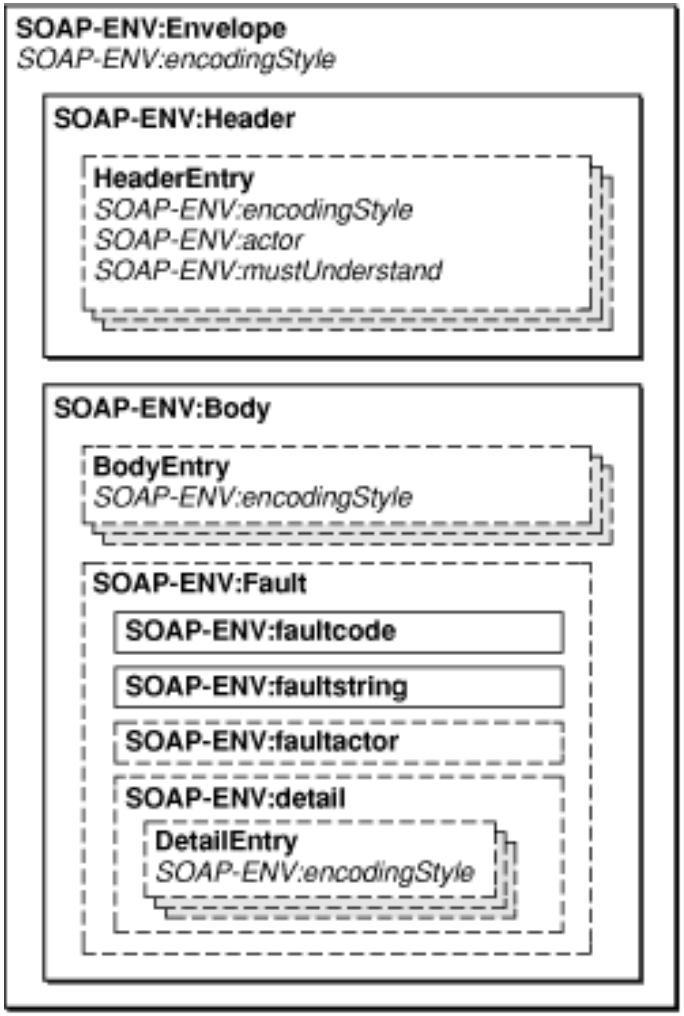
- Provides mechanisms to:
  - description of the communication unit of the SOAP message;
  - error handling;
  - representation of the data;
  - remote procedure call;
  - connection with HTTP.

SOAP Architecture



The elements of a SOAP message

- The Envelope is the root element of the message.
- Header - not a mandatory element of the message. May contain additional information for the application that processing the request.
- Body - the obligatory element of the message. Contains calls to the methods and parameters passed.



SOAP MESSAGE TEMPLATE

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap=http://www.w3.org/2001/12/soap-envelope
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Header>
  ...
  ...
</soap:Header>
<soap:Body>
  ...
  ...
  <soap:Fault>
    ...
    ...
  </soap:Fault>
</soap:Body>
</soap:Envelope>
```

W3C schools website (http://www.w3schools.com)

An example of the SOAP header

- In the title, we can introduce a new element not covered by SOAP standard. For example, the number of the transaction.
- Attributes:
  - mustUnderstand - the recipient must process this element;
  - actor - specifies a specific destination application to process a message in the thread.

```
<soap:Header>

   <trans:Transaction
   xmlns:trans="http:
   //www.host.com/namespaces/space/"
   soap:mustUnderstand="1">

   12

   </trans:Transaction>

</soap:Header>
```

The body of the SOAP message

Request

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetails xmlns="http://warehouse.example.com/ws">
       <productID>12345</productID>
     </getProductDetails>
   </soap:Body>
 </soap:Envelope>
```

Response

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
       <getProductDetailsResult>
         <productID>12345</productID>
         <productName>A cup</productName>
          <description>A  cup. 200 ml. </description>
         <price>9.95</price>
         <inStock>true</inStock>
       </getProductDetailsResult>
     </getProductDetailsResponse>
   </soap:Body>
 </soap:Envelope>
```

SOAP AND HTTP BINDING

- SOAP message transfer occurs over HTTP protocol via a POST request (starting with SOAP 1.2 standard, you may apply the GET request)
- Standard communication protocol:
  - The client sends a request
  - The server responds with OK

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type:
application/soap+xml;
charset=utf-8
Content-Length: nnn
…

HTTP/1.1 200 OK
Content-Type:
application/soap; charset=utf-
8
Content-Length: nnn
…
```