XML - RPC

Sara Drago

Università degli Studi di Genova

drago@disi.unige.it

Introduzione

La comparsa di XML-RPC risale al 1998, ad opera di Dave Winer di UserLand Software.

Winer lavorava a uno dei problemi classici della computazione distribuita: come far comunicare il software tra diverse piattaforme?

Molti erano già gli standard per l'invocazione remota di procedure e metodi:

- CORBA, un noto framework per la gestione di oggetti distribuiti, utilizza l'*Internet Inter-ORB Protocol* (IIOP).
- DCOM, di Microsoft, utlizza l'*Object Remote Procedure Call* (ORPC).
- Java, per la *Java Remote Method Invocation* (RMI) utilizza il *Java Remote Method Protocol* (JRMP).

Perchè XML-RPC

- Parallelamente, si stava affermando la convinzione che si potesse trarre vantaggio dall'ampia infrastruttura offerta dal Web per rendere pubbliche informazioni utili a programmi diversi dai browser.
- In particolare, si voleva orientare verso questa prospettiva una tecnologia ben nota e molto più vecchia del Web: le Remote Procedure Call
- L'intuizione fu ignorare completamente le difficoltà dovute al trasporto, delegandole all' HTTP, per focalizzare invece l'attenzione sul contenuto dei messaggi di `POST`.

  L' XML, venuto fuori più o meno in quegli stessi anni, sembrava lo strumento ideale.

  Il problema diventava definire una grammatica XML che specificasse il nome del codice da eseguire in remoto e tutti i parametri di cui il codice potesse avere bisogno.

HTTP

- Quando l'HTTP comparve per la prima volta, era molto semplice: i client potevano fare delle richieste a un server sulla porta 80, inviando una semplice richiesta:

```
GET /docs/mydoc.html
```

- Il server rispondeva inviando un file HTML come testo ASCII e chiudendo la connessione TCP/IP. I messaggi di errore venivano riportati semplicemente come testo, restituito al client.

HTTP 1.0 e POST

- La revisione principale dell' HTTP, la 1.0, portò all'introduzione di metodi più sofisticati di `GET`.
- Anche una semplice richiesta GET ha delle informazioni di identificazione (sia in richiesta che in risposta) che non esistevano nella versione precedente
- Tra i nuovi metodi il più importante è `POST`, che fornisce un supporto per l'invio di informazioni dai browser degli utenti al server ben più complicate di un percorso del filesystem (form)

Esempio:Listener

```
import java.net.*;
import java.io.*;
public class test
 { public static void main (String[] asArgs)
   { try
      {ServerSocket ss=new ServerSocket(1234);
       Socket s=ss.accept();
       InputStream is=s.getInputStream();
       while(true)
         {int i=is.read();
         if (i==-1) break;
         System.out.write(i);
         }
       System.out.println("\n\nConnection closed");
       }
     catch (IOException ioe)
       {
        System.err.println("Exception:");
        ioe.printStackTrace();
       }
   }
}
```

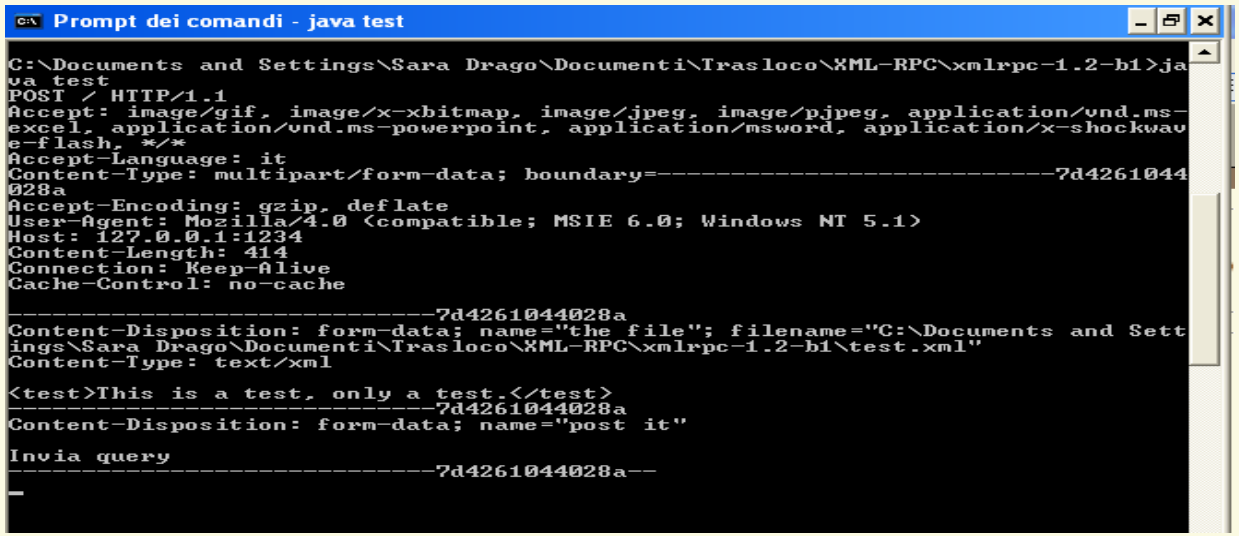
Es: form che usa il POST

```
<html>
<head><title>Testing POST - files</title>,</head>
<body>
<form method="POST"
   action="http://127.0.0.1:1234"
   enctype="multipart/form-data">
<input name="the file" type="file"><br>
<input name="post it" type="submit">
</form>
</body></html>
```

Es:Upload

test.xml

```
<test>This is a test, only a test.</test>
```



XML

- XML sta per E**X**tensible **M**arkup **L**anguage
- XML è un linguaggio di markup molto simile all' HTML
- XML è progettato per descrivere i dati
- XML non ha tag predefiniti, ma è l'utente a definirli

XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
        <to>Tove</to>
        <from>Jani</from>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend!</body>
</note>
```

- **Tutti gli elementi XML devono avere un tag di chiusura**
- **I tag XML sono case sensitive**
- **Ogni documento XML deve avere un elem. radice**
- **Gli elementi sono innestati gerarchicamente**
- **Gli elementi possono avere attributi**

Validità di un documento XML

- Un documento XML “ben formato” è un documento sintatticamente corretto
- Un documento XML “valido” è un documento validato rispetto a un DTD
- Il DTD (Document Type Definition) definisce gli elementi legali di un documento XML
- Lo Schema è un modo alternativo di definire gli elementi accettati

Validità e Parsing

- **Un elaboratore XML convalidante deve confrontare il documento che gli viene passato confrontandolo con la sua DTD.**
- **Internet Explorer 5.0 può validare un documento rispetto alla sua DTD.**
- **Per leggere, aggiornare, creare e manipolare un documento XML, è necesssario un parser XML.**

Tipi di Parser

- I parser XML sono librerie di codice che si interfacciano con i programmi: si invia l'XML al parser e si ottengono info sul contenuto
- **Parser ad eventi (SAX):** il parser chiama una funzione del programma quando si incontra un evento, es l'inizio o la fine di un elemento
- **Parser basati su modelli a oggetti dei documenti (DOM)** leggono l'intero documento e restituiscono una struttura ad albero che si puo` interrogare o modificare

Punti di forza di XML-RPC

Cosa comporta basare un protocollo su XML?

- Tutti i protocolli citati all' inizio sono binari, mentre **XML è testuale**: i dati possono essere condivisi indipendentemente dall'hardware e dal software
- Il **debugging è notevolmente semplificato**, perché l'XML è leggibile anche da essere umani.
- I dati sono molto più **firewall-friendly**: un firewall può analizzarli e dedurre che sono innocui, facendoli passare.

Coreografia di XML-RPC

Gli attori sono due: un *client* (il processo che fa la chiamata) ed un *server* (il processo chiamato). Il server viene reso disponibile ad un URL particolare.

I passi del client:

1. Il programma client chiama una procedura usando il client XML-RPC. Devono essere specificati il nome di un metodo, i parametri e il server di destinazione.

2. Il client XML-RPC
   - ✓ impacchetta il nome del metodo e i parametri come XML;
   - ✓ invia al server una richiesta di `POST` HTTP contenente le informazioni sulla richiesta.

Coreografia di XML-RPC

I passi del server:

1. Un server HTTP sulla macchina di destinazione riceve la richiesta `POST` e passa il contenuto XML ad un listener XML-RPC
2. Il listener fa il parsing dell'XML per avere il nome del metodo e i parametri. Chiama poi il metodo appropriato, passandogli I parametri.
3. Il metodo restituisce una risposta, che viene impacchettata come XML.
4. Il server Web restituisce tale XML come risposta alla richiesta `POST` HTTP.

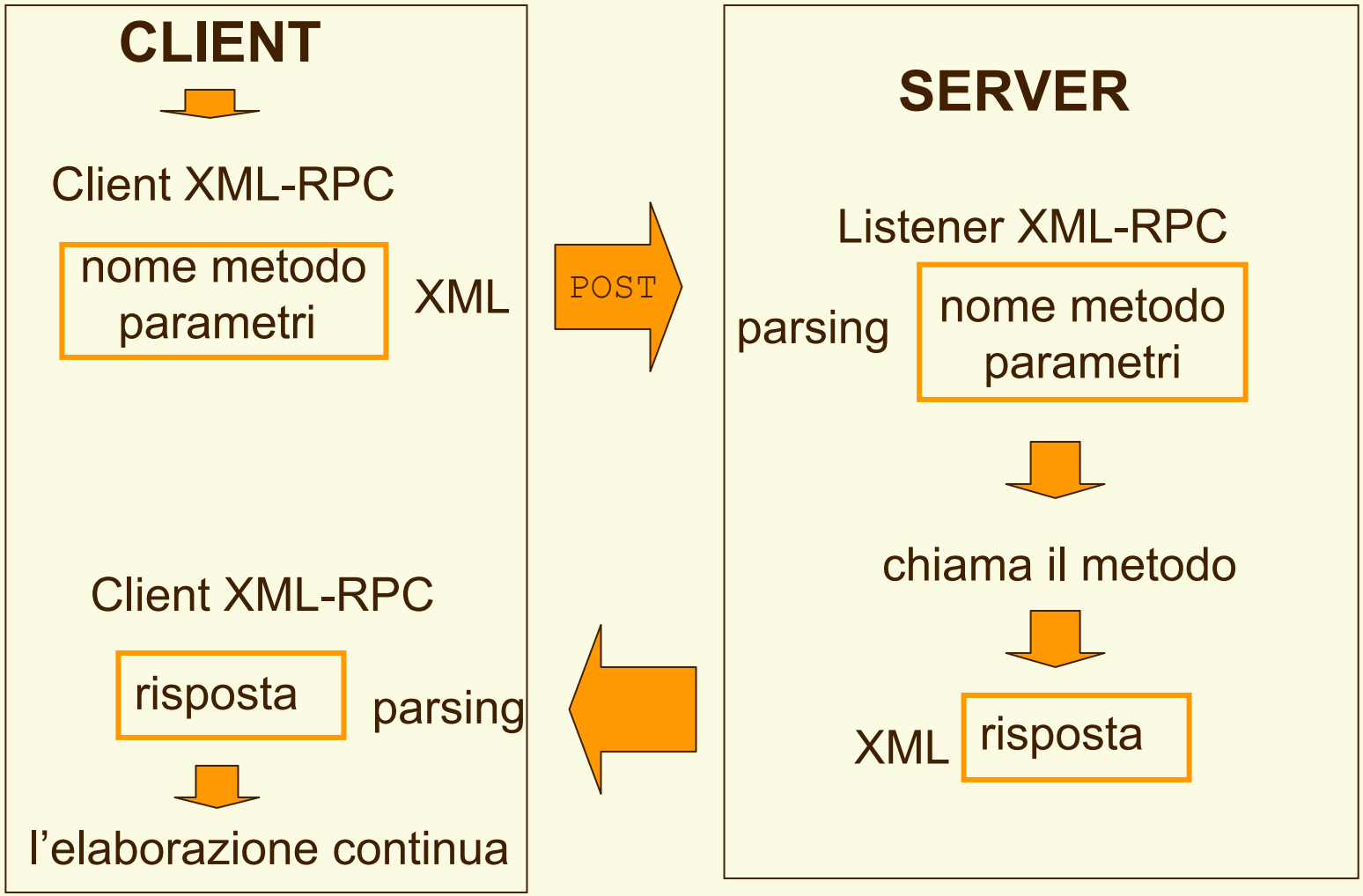
Coreografia di XML-RPC

Torna in scena il client:

1. Il client XML-RPC fa il parsing dell' XML per estrarre il valore di ritorno e lo passa al programma client;
2. Il programma client elabora il valore restituito e continua le sue operazioni.

Oss: E' possibile che un processo agisca sia da client che da server, inviando e ricevendo richieste XML-RPC. E' però sempre possibile, all'interno di ciascuna richiesta, riconoscere i due ruoli.

Schema di Riepilogo



Uso dell'HTTP: Conseguenze

L'uso dell'HTTP implica che le richieste XML-RPC debbano essere sia sincrone che stateless.

Sincrone

Una richiesta XML-RPC è sempre seguita da una e una sola risposta sincrona con la richiesta stessa. Questo accade perchè la risposta deve avvenire sulla stessa connessione HTTP della richiesta.

Il processo client resta in attesa finchè non riceve una risposta (il codice dovrebbe dunque essere progettato in modo che il blocco di una risposta non influisca troppo sulle operazioni).

E' possibile implementare sistemi *asincroni*, in cui la risposta ad una richiesta è inviata in un momento successivo, ma è molto più complicato e non sempre necessario.

Uso dell'HTTP: Conseguenze

Stateless

Nell' HTTP da una richiesta all'altra non viene conservato il contesto.

Questo vuol dire che se il client invoca un metodo sul server e poi lo invoca di nuovo, le due chiamate costituiscono due eventi isolati e non collegati.

Di conseguenza, neanche l'XML-RPC fornisce un supporto per mantenere lo stato.

E' possibile tuttavia implementare un sistema di identificatori di sessione: le procedure dovrebbero avere un database che tenga traccia di quali chiamate provengono da dove ed usare la storia delle chiamate precedenti per determinare le risposte a quelle correnti.

Il formato delle richieste

L'XML-RPC definisce una richiesta inviata ad un server per fare in modo che quest'ultimo faccia un'azione.

Tale richiesta ha due parti:

- Header HTTP (identifica il server)
- Contenuto XML (contiene le informazioni sul metodo da invocare)

Header HTTP

```
POST /rpchandler HTTP/1.0
User-Agent: AcmeXMLRPC/1.0
Host: xmlrpc.esempio.com
Content-Type: text/xml
Content-Length: 165
```

URL relativo al server dello script che deve ricevere i dati

Nome del server che servirà la richiesta **

Campo costante all'interno dell' XML-RPC.

Numero di byte nel corpo del messaggio XML-RPC, cioè tutto quello che si trova dopo la linea vuota che segue l'header

** L' header `Host` permette l'uso di un server virtuale che condivida lo stesso indirizzo IP di altri programmi di server. In alcuni casi è necessario specificare il cammino della porzione di filesystem dove si trova lo script di XML-RPC.

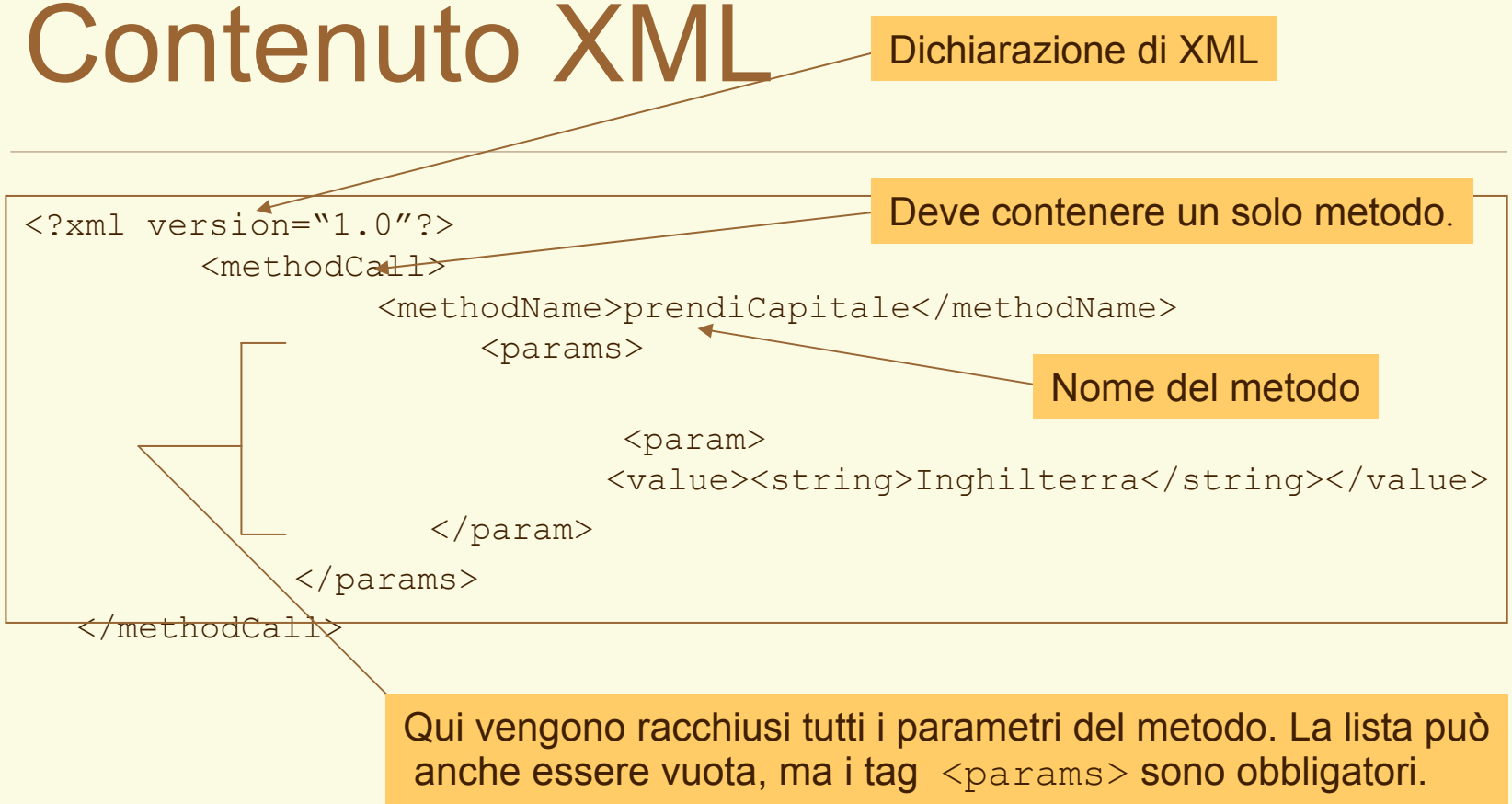
Contenuto XML

Le chiamate XML-RPC somigliano moltissimo alle chiamate a funzione dei linguaggi di programmazione.

L'invocazione di un metodo remoto è accompagnata da dati sotto forma di parametri, e la risposta contiene dei dati come valore restituito.

La rappresentazione di tali dati è in XML.

Contenuto XML




```
<?xml version="1.0"?>
        <methodCall>
                <methodName>prendiCapitale</methodName>
                     <params>
                              <param>
                             <value><string>Inghilterra</string></value>
                        </param>
                </params>
    </methodCall>
```

- Ciascun parametro è un valore XML-RPC e deve essere racchiuso tra i tag `<param>…</param>`
- Non si possono creare metodi che accettano un numero variabile di parametri.

Valori XML-RPC

Qualsiasi elemento di dati in una richiesta o risposta XML-RPC è contenuto in un elemento `<value>…</value>`.

Tipi di dati semplici

Interi `<int>…</int>` opp. `<i4>…</i4>`

Numeri in virgola mobile `<double>…</double>`

Valori booleani `<boolean>…</boolean>`

Stringhe default opp. `<string>…<string>`

Date e ore `<dateTime.iso8601>…</dateTime.iso8601>`

Binari `<base64>…</base64>`

Valori XML-RPC

Tipi di dati strutturati

Array `<array>…</array>`

```
<value>
 <array>
  <data>
   <value> valore XML-RPC </value>
     …
   <value> valore XML-RPC </value>
  </data>
 </array>
</value>
```

Oss: Gli array XML-RPC si possono considerare delle liste non tipate, perché gli elementi non sono necessariamente dello stesso tipo e non sono numerati come ci si aspetterebbe.
Un elemento di un array può essere sia semplice che strutturato.

Valori XML-RPC

Tipi di dati strutturati

Struct `<struct>…</struct>`

```
<value>
 <struct>
  <member>
   <name>Nome-1</name>
    <value> valore-1 </value>
  </member>
     …
  <member>
   <name>Nome-n</name>
    <value> valore-n </value>
  </member>
 </struct>
</value>
```

Le implementazioni di XML-RPC fanno una conversione tra le struct e i tipi di dato dizion. del linguaggio di progr. dell'host.

Una struct è una serie di campi ciascuno dei quali è una coppia (nome, valore).
Il nome deve essere una stringa ASCII, il valore un tipo di dato XML-RPC, anche strutturato.

La lista dei campi viene trattata come non ordinata.

Le specifiche non forzano i nomi ad essere distinti.

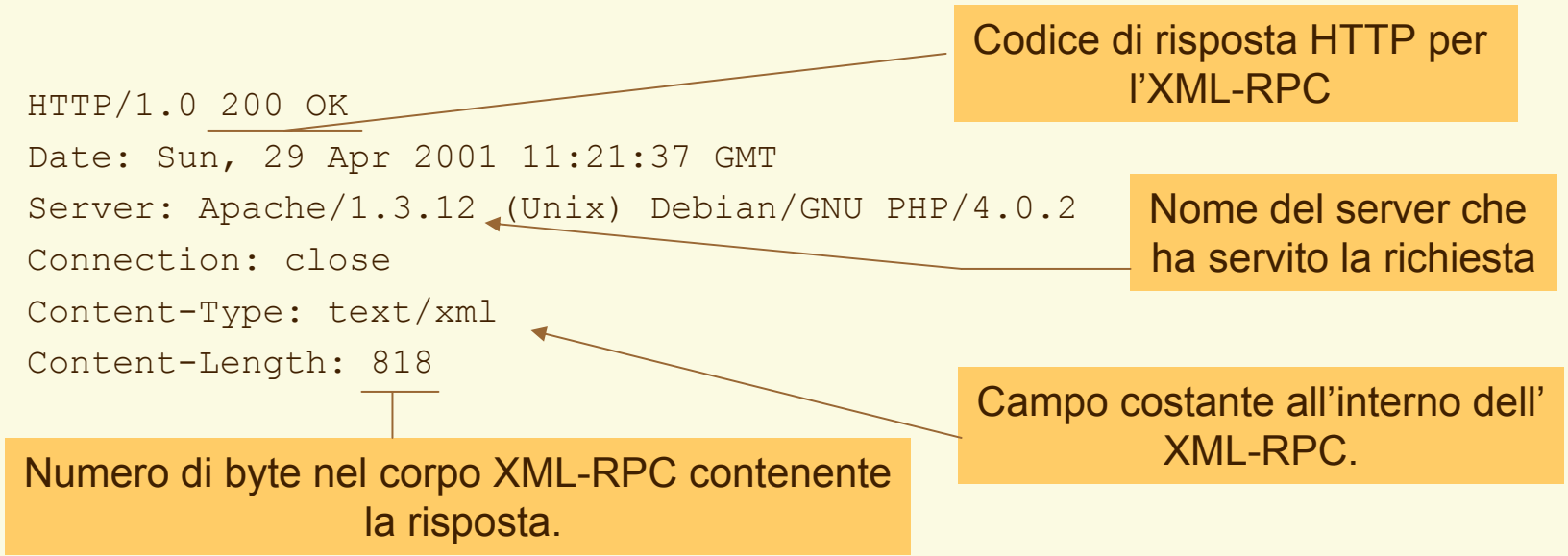
Il formato delle risposte

Dopo aver ricevuto una richiesta XML-RPC, il server deve inviare una risposta al client.

Come la richiesta, la risposta ha due parti:

- Header HTTP
- Contenuto XML (contiene il risultato del metodo o un messaggio di errore)

Header HTTP



Il fatto che l'header `Content-Length` indichi il numero di byte nella risposta significa che non è possibile creare una risposta in streaming.

Il Risultato del Metodo

```
<?xml version="1.0"?>
        <methodResponse>
           <params>
                          <param>
                 <value><string>Londra</string></value>
                 </param>
           </params>
   </methodResponse>
```

Dichiarazione di XML

Il contenuto del messaggio di risposta si configura come un elenco di parametri, ma contiene solo un `<param>`.

A volte un metodo remoto non deve restituire nessun valore, come una funzione definita `void` in Java. XML-RPC , invece, obbliga a restituire esattamente un parametro. Modi per aggirare l'ostacolo:

- restituire il booleano `true`;
- usare `Nil` (ove possibile);
- usare un valore qualsiasi, purché l'utente sappia che è privo di senso.

I Messaggi di Errore

L'XML-RPC fornisce una struttura di messaggi di errore molto simile alle eccezioni di alcuni linguaggi di programmazione moderni.

Se mentre viene gestita una richiesta avviene un errore di elaborazione, nella risposta del metodo viene inserita una struttura `<fault>…<fault>` invece del risultato dell'elaborazione.

Una struttura fault contiene un singolo valore XML-RPC, costituito da una struct con due campi.

I Messaggi di Errore

```
<?xml version="1.0"?>
<methodResponse>
 <fault>
  <value>
   <struct>
     <member>
      <name>faultCode</name/>
      <value><int>3</int></value>
     </member>
     <member>
      <name>faultString</name>
      <value>
       <string>No such method.</string>
      </value>
     </member>
    </struct>
   </value>
 </fault>
</metrodResponse>
```

Indicatore numerico dell'errore avvenuto.
Può cambiare a seconda delle implementazioni.

Spiegazione testuale dell'errore

Nei linguaggi che le supportano, i fault sono mappati sulle eccezioni. In altri linguaggi, è necessario un costrutto condizionale per controllare il valore di ritorno.

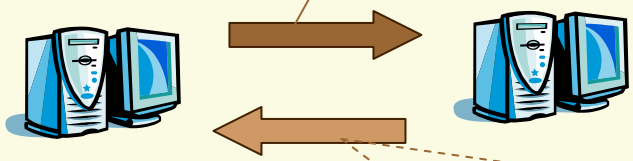
Osservazioni

- Tutto quello che è stato informalmente descritto sulla struttura del contenuto XML poteva essere detto con una DTD
- Non tutti gli strumenti XML-RPC fanno la convalida con una DTD, ma hanno della programmazione interna che controlla
- Il parser è invece necessario e nella maggior parte delle implementazioni è incluso un piccolo parser ad hoc

Esempio

```
POST/RPC HTTP/1.0
---
<?xml version="1.0"?>
<methodCall>
  <methodName>getGuaranteedDeliveryTime</methodName>
    <params>
       <param>
            <value><string>0-13-18188-222</string></value>
       </param>
       <param>
            <value><string>75240</string></value>
       </param>
    </params>
            </methodCall>
```

Richiesta del client



Risposta del server

```
HTTP/1.1 200 OK
<?xml version="1.0"?>
<methodResponse>
<params>
 <param>
  <value><string>2 hours</string></value>
</param>
</params>
</methodResponse>
```

Messaggio di errore

```
HTTP/1.1 200 OK
---
<?xml version="1.0"?>
<methodResponse>
 <fault>
  <value>
   <struct>
     <member>
      <name>faultCode</name/>
      <value><int>4</int></value>
     </member>
     <member>
      <name>faultString</name>
      <value>
       <string>Improper ISBN</string>
      </value>
     </member>
    </struct>
   </value>
 </fault>
</methodResponse>
```

XML-RPC e Java

Java è già un ambiente estremamente orientato alle reti, completo di tutti i suoi meccanismi per la comunicazione remota ed il coordinamento di oggetti su più sistemi.

Può contare su:

- RMI per la comunicazione con altri ambienti Java;
- CORBA per la connessione con sistemi non omogenei.

Tuttavia XML-RPC può presentare alcuni vantaggi:

- È più leggero di RMI perchè non è necessario il marshalling e tutti i parametri vengono tradotti in XML;
- Ha bisogno di meno risorse di CORBA.

La libreria Java XML-RPC

All'indirizzo **`http://ws.apache.org/xmlrpc/`** è disponibile per Java una libreria scaricabile gratuitamente che comprende:

- un insieme di classi per la creazione di client e server XML-RPC;
- un piccolo server XML-RPC che permette di lavorare senza il carico aggiuntivo di un Web server completo;
- un parser XML

Il pacchetto org.apache.xmlrpc supporta tutti i tipi di dati di XML-RPC (anche `nil`) rappresentandoli come tipi di dati di Java:

Tipo XML-RPC	Tipo Java sempl.	Tipo Java compl.
`i4`	`int`	`java.lang.Integer`
`int`	`int`	`java.lang.Integer`
`boolean`	`Boolean`	`java.lang.Boolean`
`string`	`java.lang.String`	`Java.lang.String`
`double`	`double`	`Java.lang.Double`
`dateTime`	`java.util.Date`	`java.util.Date`
`struct`	`java.util.Hashtable`	`java.util.Hashtable`
`array`	`java.util.Vector`	`java.util.Vector`
`base64`	`byte[]`	`byte[]`
`nil`	`null`	`null`

Configurazione Generale

- `org.apache.xmlrpc` contiene un insieme di metodi statici usati per configurare l'elaborazione, tra cui:
- `setDriver()`: permette di scegliere un parser XML ,per default la libreria usa MinML.
- `setDebug()`: permette di controllare la registrazione e le richieste di elaborazione dei metodi XML-RPC fornendo indicazioni sulla struttura del documento ricevuto

Costruire Client XML-RPC

Il procedimento si basa su tre strumenti fondamentali:

- la classe `XmlRpcClient;`
- il suo costruttore: crea il client e allo stesso tempo identifica il server

```
XmlRpcClient client=
        new
XmlRpcClient(http://192.168.126.42:8899
/)
```

- il metodo `execute()`.Genera una richiesta XML-RPC e la manda al server. Parsa il risultato e restituisce l'oggetto corrispondente.

Costruire Server XML-RPC

E' leggermente più complesso che creare client, perché oltre a costruire la logica del nucleo dell' applicazione, si devono registrare i servizi disponibili al pubblico.

Il protagonisti sono:

- la classe `WebServer;`
- la classe `XmlRpcServer`;
- i metodi `addHandler()` e `removeHandler()`.

L'approccio alla registrazione dei metodi è identico per le due classi.

La classe `WebServer`

- Fornisce il nucleo delle funzionalità HTTP usate dall' XML-RPC, rendendo possibile configurare XML-RPC sui sistemi che non hanno un server Web già installato.
- Il costruttore

```
WebServer server=new Webserver(9876)
```

- `server.setParanoid(true)`
  `server.addClient`(IP)
  `server.denyClient`(IP)

La classe `XmlRpcServer`

- Si integra con server Web già esistenti. Invia le richieste ai gestori, ma non tratta la gestione della connessione HTTP (si occupa solo dell'invio e della ricezione di XML per gestire le richieste)
- Il costruttore

  ```
  XmlRpcServer server=new XmlRpcServer()
  ```

- `execute(java.io.InputStream is)`

  Parsa la richiesta ed esegue il metodo gestore, se ne trova uno.

Metodi Comuni alle due Classi

- `addHandler(java.lang.String name, java.lang.Object target)`

  registra un oggetto gestore con questo nome.

- `removeHandler(java.lang.String name)`

  rimuove un oggetto gestore precedentemente registrato su questo server.

Creare Gestori

- Registrazione automatica:
  - Creo la classe con il metodo che voglio rendere disponibile attraverso XML-RPC
  - Registro l'oggetto con `addHandler`
- Registrazione esplicita:
  - Creo una classe che implementi l'interfaccia `XmlRpcHandler`
  - ho solo un metodo da implementare: `execute(String methodName,Vector parameters)`

Un Esempio Pratico

```
Public class AreaHandler {
        public Double rectArea(double length, double width) {
        return new Double(length*width);
        }
        public Double circleArea(double radius) {
        double value=(radius*radius*Math.PI);
        return new Double (value);
        }
}
```

Questa semplice funzione di libreria verrà eseguita come client sul server. Anche se i calcoli non sono molto complessi, l'approccio adottato nell'esempio che segue può essere utilizzato per algoritmi più impegnativi.
Il server verrà creato usando la classe `WebServer`.

Un Esempio Pratico:il Server

```
import java.io.IOException;
import org.apache.xmlrpc.WebServer;
import org.apache.xmlrpc.XmlRpc;
public class AreaServer {
            public static void main (String[] args) {
                        if (args.length < 1) {
                            System.out.println(
                                    "Usage: java AreaServer [port]");
                            System.exit(-1);
                        }
                        try {
                             System.out.println("Attempting to start XML-RPC Server...");
                             WebServer server = new WebServer(Integer.parseInt(args[0]));
                             server.setParanoid(true);
                             server.acceptClient("127.0.0.1");
                             XmlRpc.setDebug(true);

                             System.out.println("Started successfully.");

                             server.addHandler("area", new AreaHandler());
                             System.out.println(
                                    "Registered AreaHandler class to area.");
                             server.start();
                             System.out.println("Now accepting requests. " +
                                    "(Halt program to stop.)");
                                    System.in.read() ;
                                    } catch (Exception e) {
                                 System.out.println("Could not start server: " +
                                     e.getMessage());
                        }
            }
}
```

Avvia il server

Registra la classe gestore come area

Un Esempio Pratico: il Client

```
import java.io.IoException;
import java.util.Vector;
import org.apache.xmlrpc.XmlRpc;
import org.apache.xmlrpc.XmlRpcClient;
import org.apache.xmlrpc.XmlRpcException;
public class AreaClient {
 public static void main (String args[]) {
    if (args.length < 1){
       System.out.println(
         "Usage: java AreaClient [radius]"); System.exit (-1);}
    try{
      XmlRpcClient client= new XmlRpcClient("http://127.0.0.1:8899/");
      Vector params = new Vector();
      params.addElement(new Double(args[0]));
      Object result = client.execute("area.circleArea", params);
      System.out.println("L'area del cerchio è:" + result.toString());
      } catch (Exception e) {
          System.out.println(" Exception:"+ e.getMessage());
      }
 }
}
```

Crea il client, identificando il server

Crea i parametri della richiesta, usando l'input dell'utente

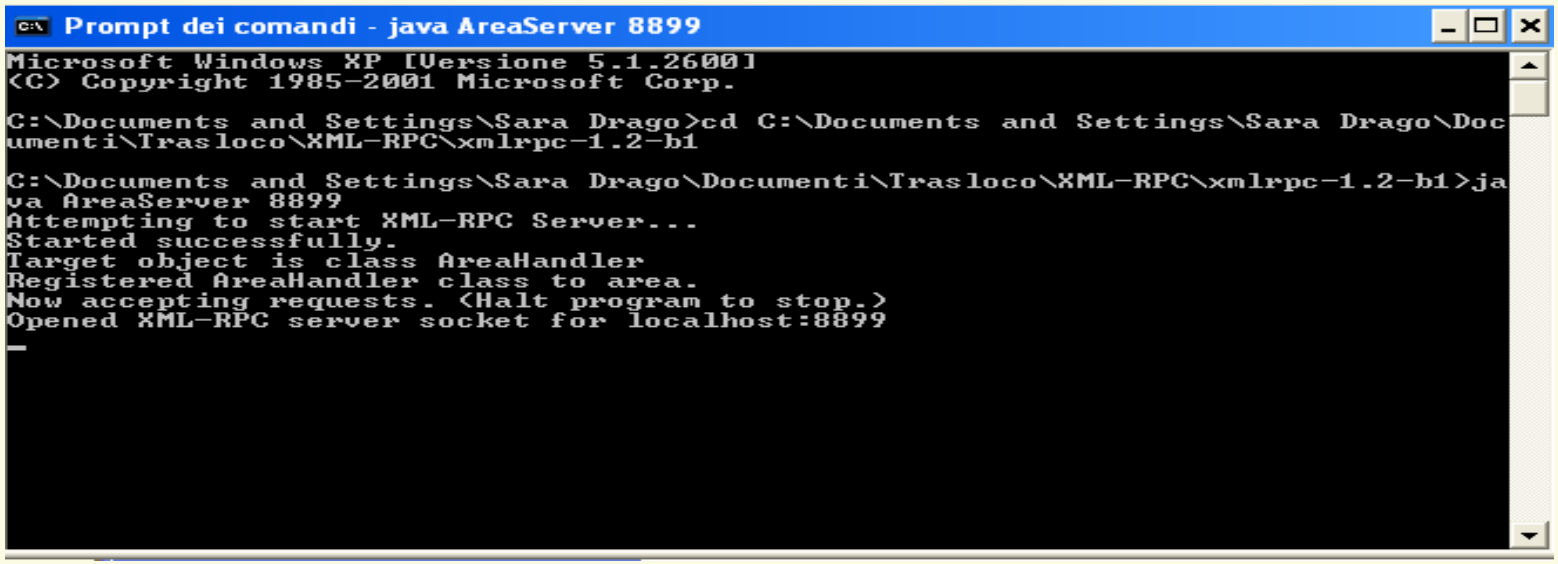
Chiama il metodo della classe gestore

Esempio

Il server si avvia con una chiamata del tipo:

C:\xmlrpc\esempio>java AreaServer 8899

8899 è la porta su cui il server starà in ascolto

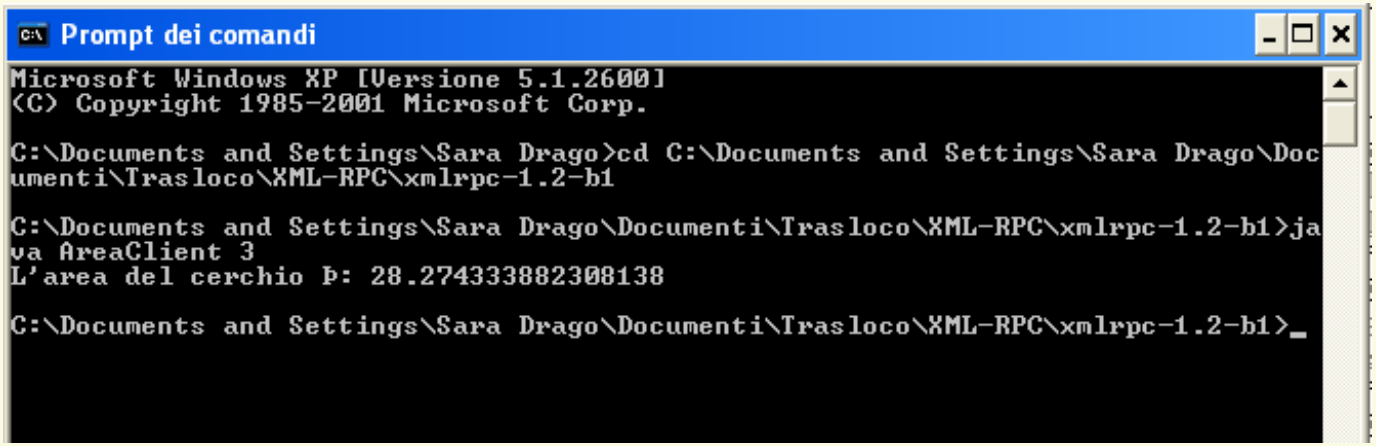


Esempio

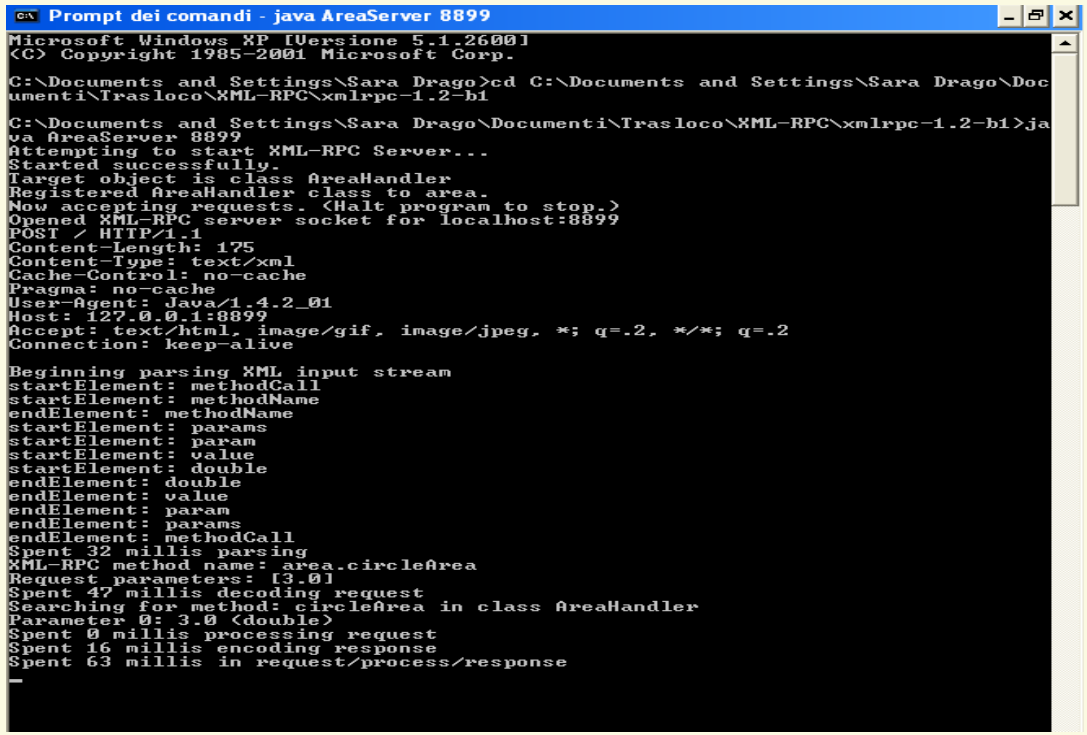
Il client è eseguito dalla linea di comando

```
C:\xmlrpc\esempio>java AreaClient 3
```

e viene data l'indicazione di un raggio (3).



Esempio



Header HTTP

Parsing del contenuto XML

Server-Richiesta HTTP

```
POST / HTTP/1.1
Content-Length: 175
Content-Type: text/xml
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Java/1.4.2_01
Host: 127.0.0.1:8899
Accept: text/html, image/gif, image/jpeg, *;
  q=.2, */*; q=.2
Connection: keep-alive
```

Server-Parsing dei dati

```
Beginning parsing XML input stream
startElement: methodCall
startElement: methodName
endElement: methodName
startElement: params
startElement: param
startElement: value
startElement: double
endElement: double
endElement: value
endElement: param
endElement: params
endElement: methodCall
Spent 0 millis parsing
```

Server-RPC

```
XML-RPC method name: area.circleArea
Request parameters: [3.0]
Spent 16 millis decoding request
Searching for method: circleArea in class
  AreaHandler
Parameter 0: 3.0 (double)
Spent 0 millis processing request
Spent 0 millis encoding response
Spent 16 millis in request/process/response
```

Esempio 2 : registr automatica

```
public class testHandler {

    public String nameTester(String first, String last) {
                return "Reversed: "+ last + ", " + first;
    }
}
```

Esempio 2: il Server

```
import java.io.IOException;
import org.apache.xmlrpc.WebServer;
import org.apache.xmlrpc.XmlRpc;
public class TestServer {
            public static void main (String[] args) {
                        if (args.length < 1) {
                            System.out.println(
                                    "Usage: java TestServer [port]");
                            System.exit(-1);
                        }
                        try {
                             //Avvia il server, usando la versione interna
                             System.out.println("Attempting to start XML-RPC Server...");
                             WebServer server = new WebServer(Integer.parseInt(args[0]));
                             server.setParanoid(true);
                             server.acceptClient("127.0.0.1");
                             XmlRpc.setDebug(true);
                             System.out.println("Started successfully.");
                             //Registra la nostra classe gestore come test
                             server.addHandler("test", new testHandler());
                             System.out.println(
                                    "Registered testHandler class to test.");
                             server.start();
                             System.out.println("Now accepting requests. " +
                                    "(Halt program to stop.)");
                                    System.in.read() ;
                                    } catch (Exception e) {
                                 System.out.println("Could not start server: " +
                                     e.getMessage());
                        }
            }
}
```

Avvia il server

Registra la classe gestore come test

Esempio 2: registrazione espl.

```
import java.util.Vector;
import org.apache.xmlrpc.XmlRpcHandler {
   public class testHandlerExpl implements XmlRpcHandler {
     public Object execute(String methodname, Vector parameters)
        throws java.lang.Exception {
           if (methodname=="nameTester") {
                String first= (String) parameters.elementAt(0);
                String last=(String) parametres.elementAt(1);
                return nameTester(first,last);
          } else {
                   throw new Exception ("No such method!");
        }
     }
     private String nameTester(String first, String last) {
        return "Reversed: "+ last + ", " + first;
    }
}
```

Esempio 2: il Client

```
import java.io.IOException;
import java.util.Vector;
import org.apache.xmlrpc.XmlRpc;
import org.apache.xmlrpc.XmlRpcClient;
import org.apache.xmlrpc.XmlRpcException;
public class TestClient {
            public static void main(String args[]) {
                        if (args.length < 1) {
                                    System.out.println(
                                                "Usage: java TestClient [string,string]");
                                    System.exit(-1);
                        }
                        try {
                            // Crea il client, identificando il server
                            XmlRpcClient client= new XmlRpcClient("http://127.0.0.1:9876/");

                            //Crea i parametri della richiesta usando l'input dell'utente
                            Vector params = new Vector();
                            params.addElement(new String(args[0]));
                            params.addElement(new String(args[1]));
                            //Invia una richiesta
                            Object result =
                                    client.execute("test.nameTester", params);
                        //Comunica i risultati
                            System.out.println("L'inversione di stringhe: " +
                                    result.toString());
                           } catch (Exception e) {
                            System.out.println("Exception: " + e.getMessage());
                           }
                        }
}
```

Crea il client, identificando il server

Chiama il metodo della classe gestore

Crea i parametri della richiesta

Limiti di XML-RPC

- Poiché XML-RPC viene dalla tradizione della programmazione procedurale, c'è poca flessibilità nei confronti di una progettazione orientata agli oggetti.
- L'HTTP era stato progettato per un utilizzo a base ristretta (il trasferimento di documenti HTML dai server ai browser): non è pensabile che garantisca l'approccio più efficiente.
- XML-RPC fornisce pochissima sicurezza per le sue transazioni e le sue capacità di superare i firewall creano ulteriori pericoli.

Oltre XML-RPC

Anche se pensiamo che XML-RPC sia un insieme di strumenti molto utile, diversi protocolli usano l'XML per trasferire le informazioni tra computer (spesso utilizzando l'HTTP).

Questa molteplicità di approcci si deve all'intenso interesse degli sviluppatori per caratteristiche in grado di superare quelle offerte dalle chiamate a procedura.

Ecco alcune delle opzioni che possono essere considerate invece dell' XML-RPC:

- SOAP (Simple Object Access Protocol)
- UDDI (Universal Description, Discovery and Integration)
- WSDL (Web Services Description Language)
- BXXP (Blocks eXtensible eXchange Protocol, non è costruito sull' HTTP, ma sul TCP)

Simple Object Access Protocol (SOAP)

Sara Drago

Università degli Studi di Genova

drago@disi.unige.it

SOAP

SOAP è un protocollo leggero che permette di scambiare informazioni in ambiente distribuito:

- SOAP è nato dallo stesso lavoro che ha generato originariamente XML-RPC
- SOAP è **basato su XML**
- SOAP gestisce **informazione strutturata e tipata**.
- SOAP non definisce semantiche per i dati e le chiamate, ma fornisce agli sviluppatori i mezzi per farlo (con un intenso uso dei *Namespace* XML, SOAP permette agli autori dei messaggi di dichiararne la semantica usando grammatiche XML definite per lo scopo in particolari namespace).

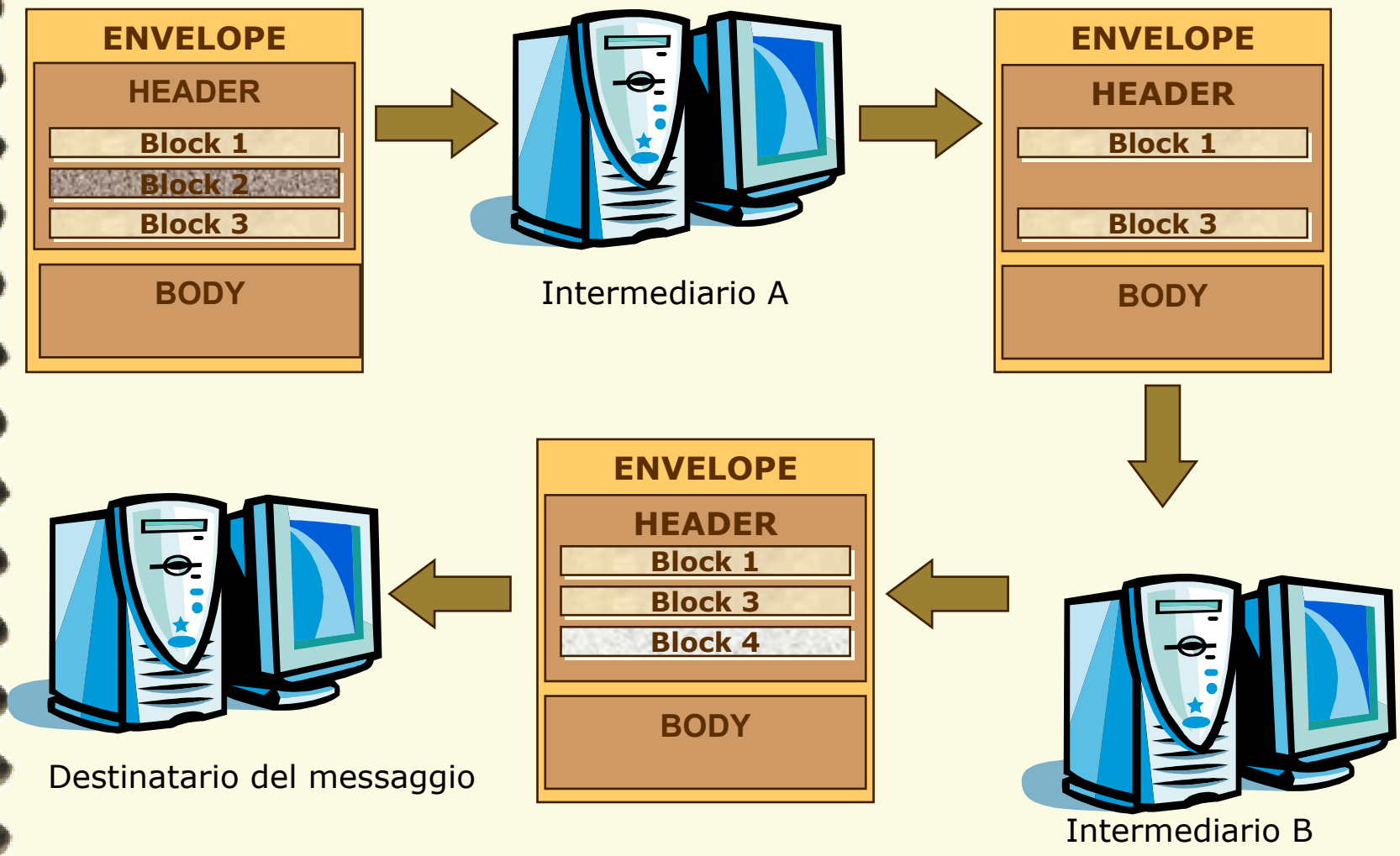
Applicazioni SOAP

- Parlare di SOAP solo in ambito RPC è in realtà errato.
- **SOAP in effetti è un generico framework per lo scambio di informazioni**.
- Nella specifica di SOAP redatta dal W3C è però inserita una sezione in cui viene descritto un meccanismo standard per la codifica delle informazioni RPC con SOAP.

SOAP Processing Model

- Un messaggio SOAP può attraversare varie applicazioni prima di raggiungere la sua destinazione.
- Gli header block non sono necessariamente diretti al destinatario finale, ma possono essere indirizzati ad applicazioni intermedie. Questo avviene tramite l'attributo `role`, che è usato per indirizzare l'header block a nodi che operano in quello stesso ruolo.
- Il nodo destinatario di un block deve sempre rimuoverlo prima di inoltrare il messaggio al nodo successivo lungo il path che conduce al destinatario.
- Un intermediario può inserire altri header block nel messaggio, ma non può toccarne il body.

SOAP Processing Model



Ragioni del Processing Model

SOAP prevede la possibilità di indirizzare diverse parti dello stesso messaggio a destinatari diversi.
Gli intermediari sono applicazioni che possono processare parti di un messaggio SOAP mentre questo si sposta dal suo punto d'origine alla destinazione.
Quali sono i vantaggi offerti da questo modello?

- Rende possibile aggiungere servizi lungo il percorso del messaggio;
- facilita l'implementazione della sicurezza, perché consente di far passare il messaggio in domini affidabili e conosciuti;
- rappresenta un superamento dell'architettura client-server, aumentando l'efficienza in numerose situazioni (es: smistamento della po -sta)
- facilita la ricostruzione dell'esatto cammino attraversato da un messaggio, consentendo di verificare la presenza di bottlenecks.

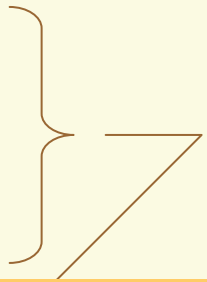
Protocolli di trasporto

SOAP è indipendente dal protocollo di trasporto e permette lo scambio di messaggi attraverso tutta una gamma di protocolli.

Il punto chiave nel decidere quale protocollo di trasporto legare a SOAP consiste nell'identificare come i requisiti che ci aspettiamo dal Web Service che stiamo realizzando vengano interpretati dalle caratteristiche del protocollo in questione.

Protocolli che possono essere legati a SOAP:

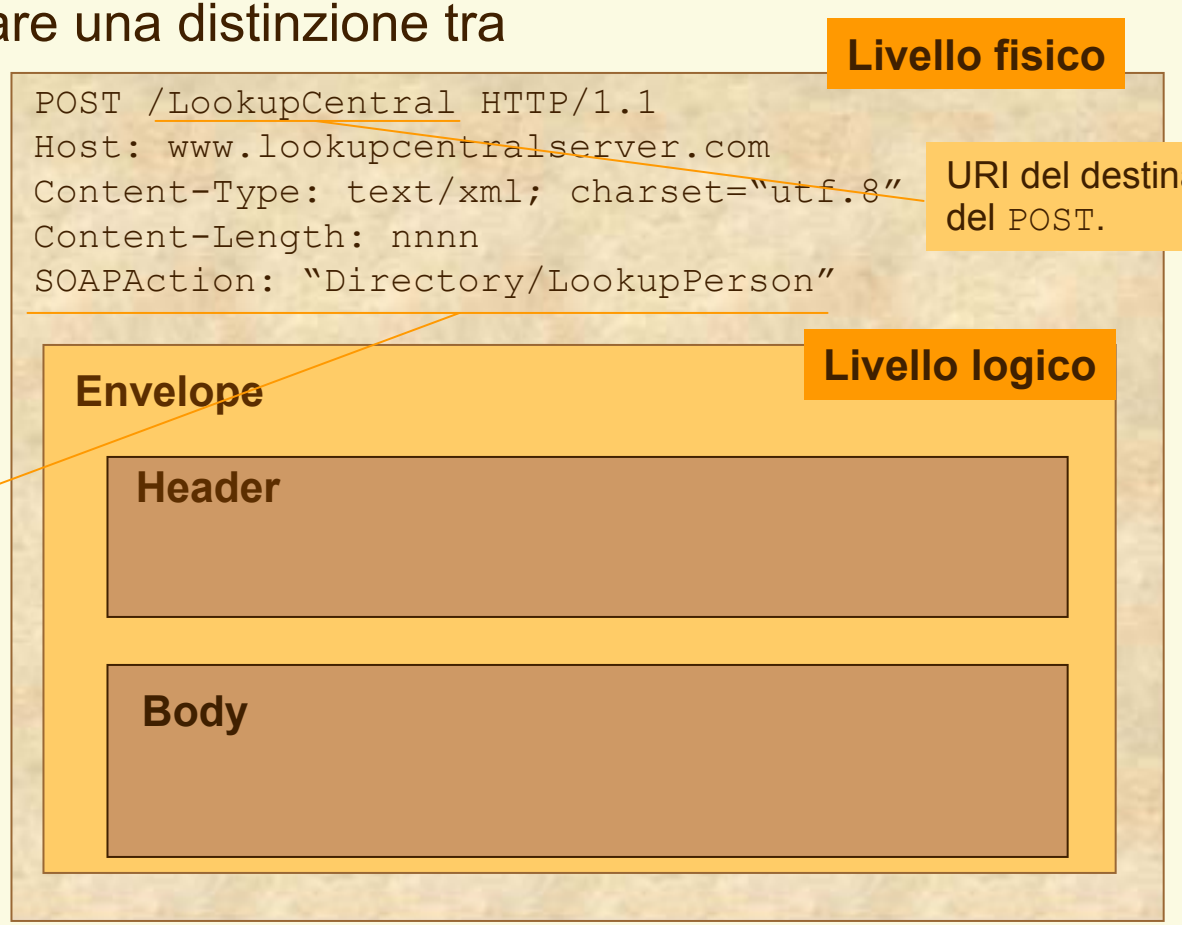
- HTTP
- HTTPS (per scambi sicuri di messaggi)
- MIME (per trasmettere messaggi con attachments)
- SMTP
- JMS, POP, IMAP, FTP...



I più usati con SOAP (offrono maggiori garanzie di interoperabilità)

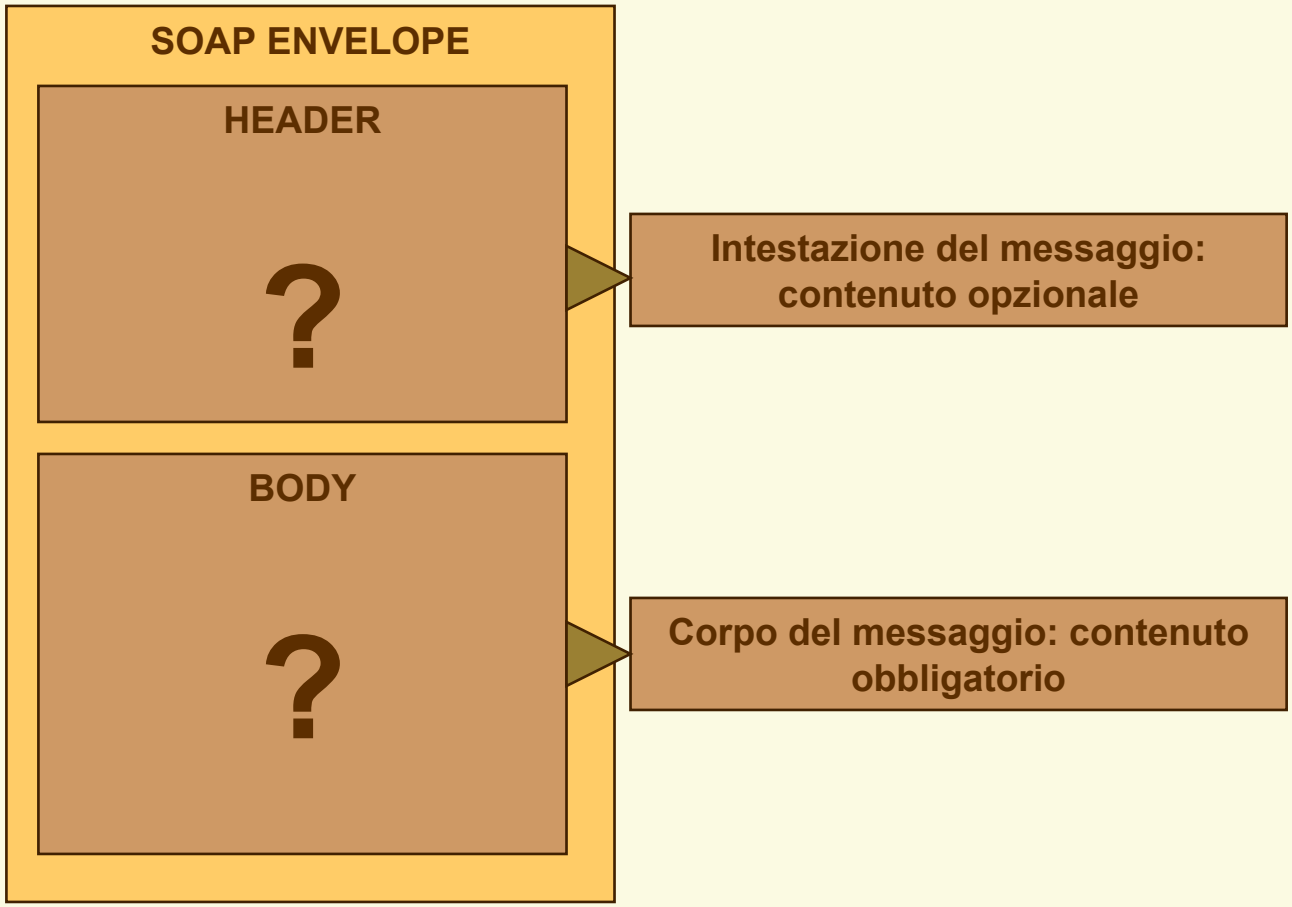
Struttura di un Messaggio SOAP

Molti protocolli hanno una nozione di incapsulamento, che rende possibile effettuare una distinzione tra livello fisico e livello logico del messaggio.



Campo obbligatorio se il payload è un messaggio SOAP. Contiene un URI (anche vuoto) che dovrebbe fornire informazioni riguardanti il contenuto del messaggio allegato.

Struttura di un Messaggio SOAP: Livello Logico



Envelope

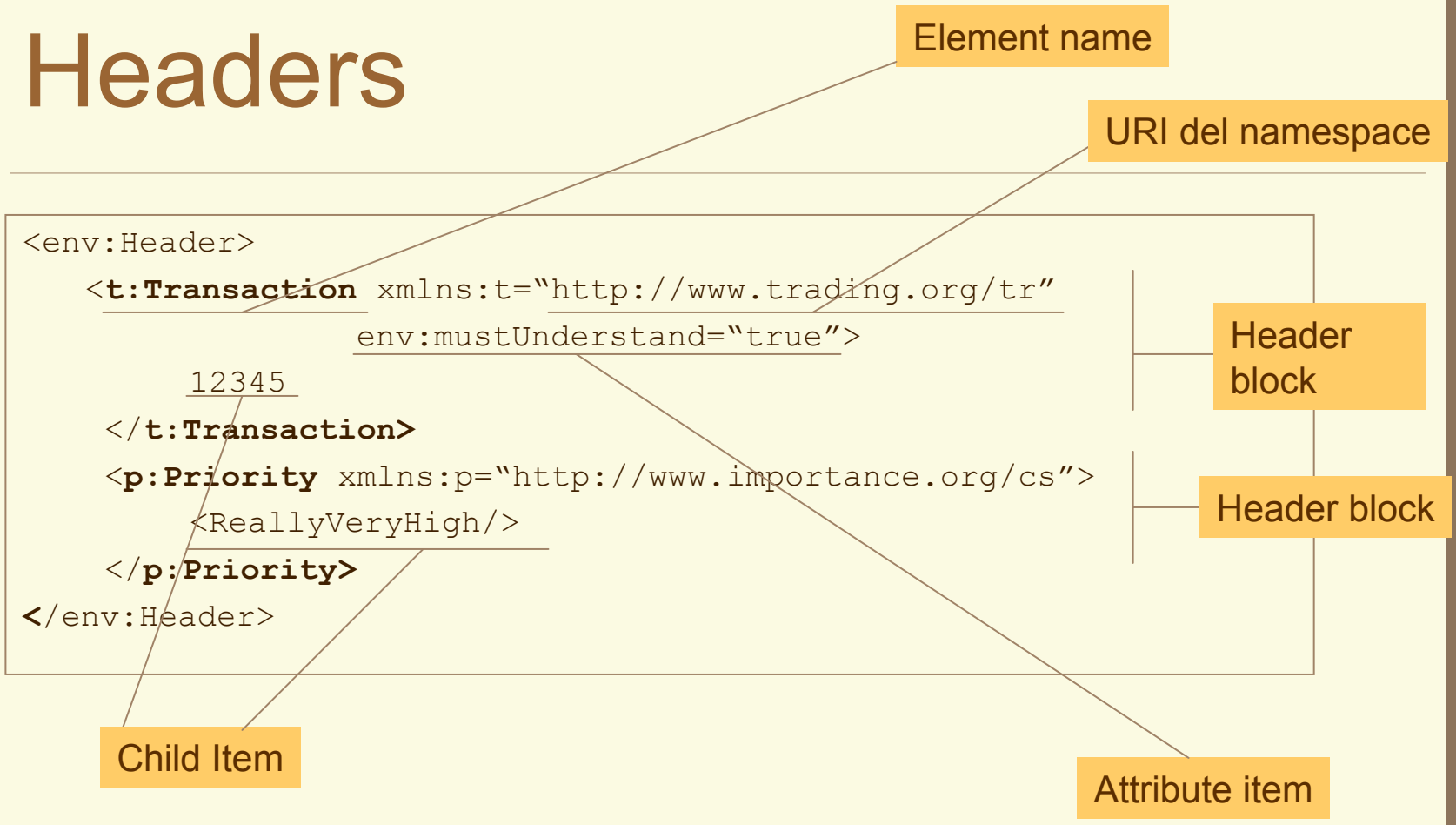
```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope"
   env:encodingStyle ="http://www.w3.org/2003/05/soap-encoding">
        ...
</env:Envelope>
```

Attributo opzionale. L'URI punta al sistema di serializzazione standard di SOAP. Se manca o se il percorso termina con …/`none`, nessuna assunzione è fatta relativamente all encoding style dell' elemento `Envelope`.

Nella versione 1.1 di SOAP si trovano namespace di `http://schemas.xmlsoap.org/soap/envelope/`

`Envelope` è l'elemento più esterno del documento SOAP ed è obbligatorio. Deve sempre essere associato a un namespace come quello indicato. Se il messaggio è ricevuto da un'applicazione associata ad un diverso namespace, si solleva un errore.

Headers



Nota: I figli dell'header sono costruiti usando grammatiche specificate dall'utente. Il loro significato sarà noto solo a chi conosce il namespace di definizione

Headers: quali Attribute Items?

- mustUnderstand

```
env:mustUnderstand="true"
```

Specifica se il destinatario dell'header block debba elaborarlo ("`true`") oppure possa
ignorarlo ("`false`") .
Omettere questo Attribute Item equivale a includerlo con il valore "`false`".

- role

```
env:role="http://www.w3.org/2003/05/soap-envelope/…"
```

L' URI esprime il ruolo dell'attore ricevente il messaggio:

`next`
`none`
`ultimateReceiver` (è il ruolo di default)

Altri ruoli sono definibili dall'utente per applicazioni particolari.

Headers: quali Attribute Items?

- relay

```
env:relay="true"
```

Specifica se il destinatario dell'header block debba ritrasmetterlo ("`true`") nel caso in cui non sia stato processato.

Omettere questo Attribute Item equivale a includerlo con il valore "`false`".

- encodingStyle

La sintassi è quella già vista per gli attributi di `Envelope`.

Osserviamo che lo scope di questo attributo, se inserito nell'header block, è quello dell'header block e di tutti i suoi figli, a meno che uno di essi non rechi un attributo di

encoding suo proprio.

Attribute Items e Comportamento dei Nodi

Role	MustUnderstand	forwarded
`next`	Yes	No, unless reinserted
	No	No, unless `relay=true`
`ultimateReceiver`	Yes	---
	No	---
`none`	---	Yes

Body

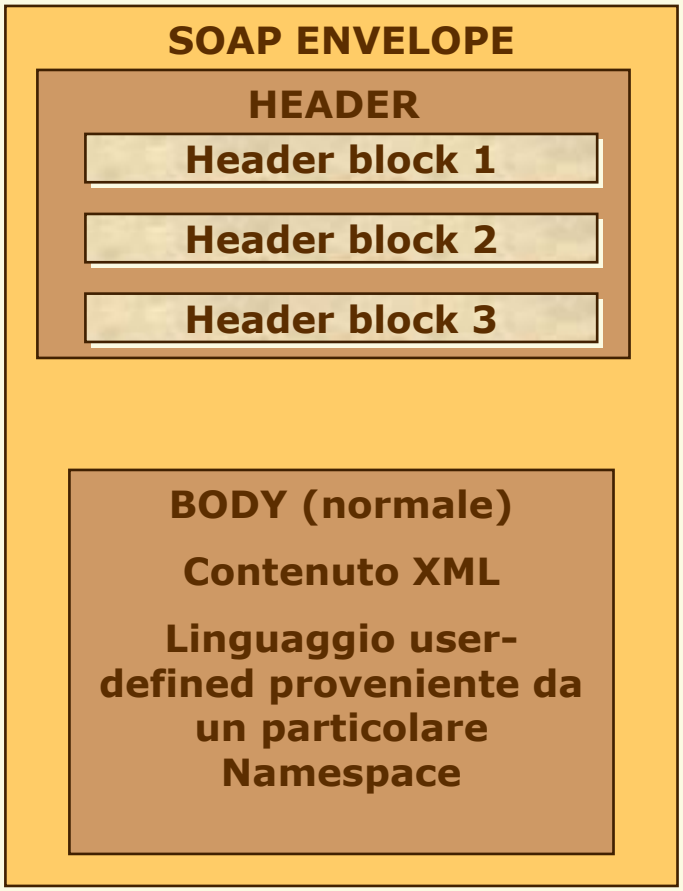
```
<env:Body>
   <m:GetLastTradePrice xmlns:m="http://www.trading.org/ex">
     <symbol>DIS</symbol>
    </m:GetLastTradePrice>
</env:Body>
```

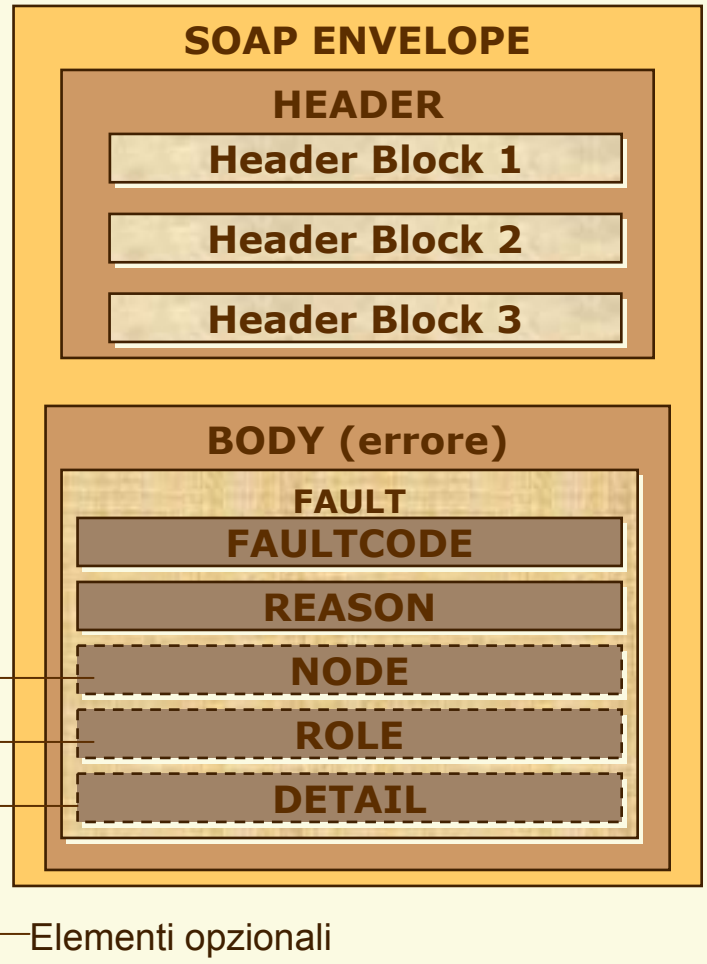
Possibilità di inserire attributi

Namespace di definizione dell'elemento figlio. Non obbligatorio, ma fortemente incoraggiato per una corretta interpretazione dei dati da parte del destinatario.
L'elemento figlio può a sua volta avere figli. A meno di ridichiarazioni, ereditano il namespace paterno.

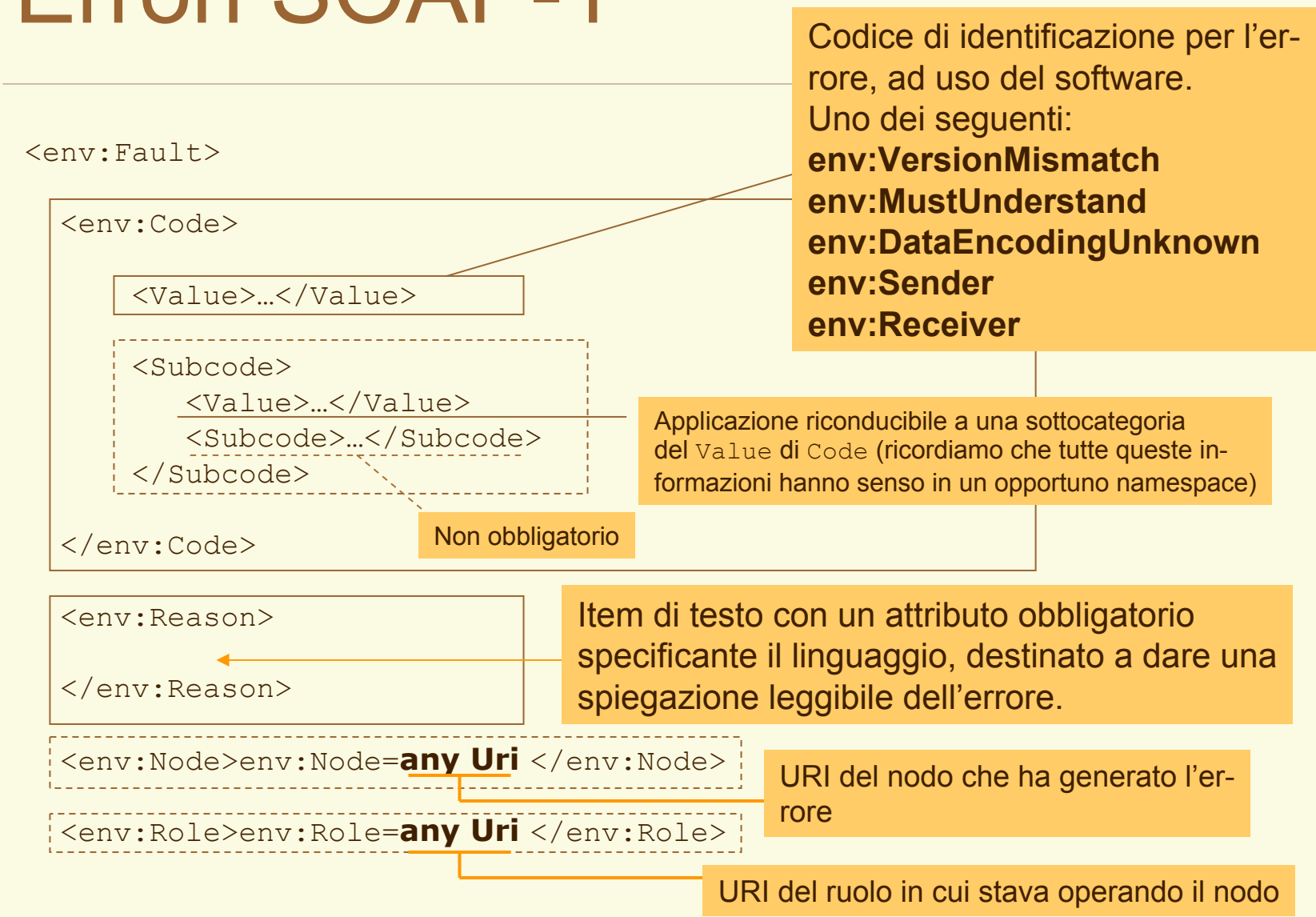
`Body` è l'elemento che racchiude le informazioni al centro del documento SOAP. Il suo contenuto è XML.
Il corpo SOAP è semanticamente equivalente a un header block con attributo `role=ultimateReceiver` e `mustUnderstand=true`.

Body


SOAP ENVELOPE
HEADER
Header block 1
Header block 2
Header block 3
BODY (normale)
Contenuto XML
Linguaggio user-defined proveniente da un particolare Namespace


SOAP ENVELOPE
HEADER
Header Block 1
Header Block 2
Header Block 3
BODY (errore)
FAULT
FAULTCODE
REASON
NODE
ROLE
DETAIL
Elementi opzionali

Errori SOAP-1



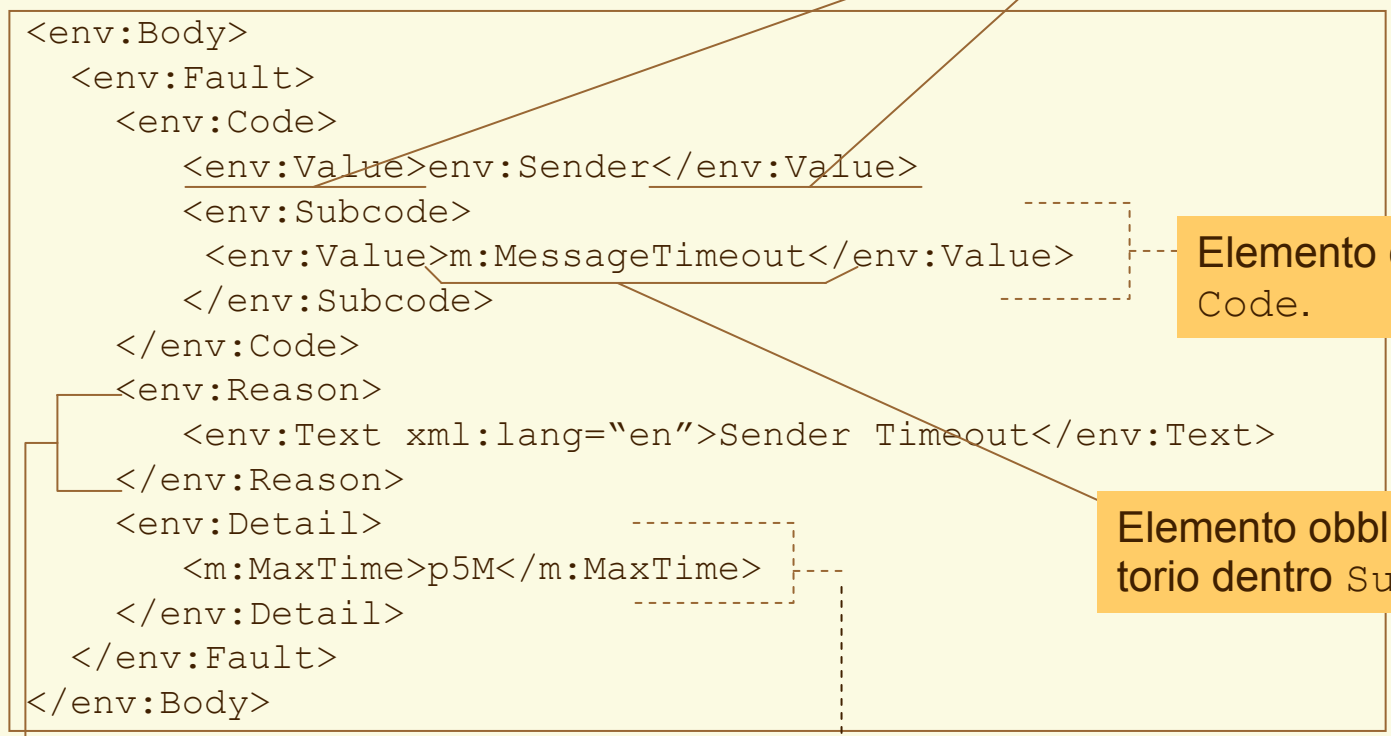
Errori SOAP-2

```
<env:Detail>…</env:Detail>
</fault>
```

Apporta informazione in più che va pensata in relazione al codice che descrive l'errore.
Può avere:
- uno o più attributi;
- uno o più elementi figli, ciascuno con un suo nome (eventuali attributi di encodingStyle);
- un elemento figlio può contenere caratteri o altri figli.

Quando è presente, l'elemento `fault` è figlio di `Body` e può comparire una volta sola. Serve a fornire informazioni su errori derivanti dall'elaborazione del messaggio.

Errori SOAP



Elemento obbligatorio all'interno di `Code`.

Elemento opzionale di `Code`.

Elemento obbligatorio dentro `Subcode**`.

Elemento obbligatorio all'interno di `Code`.

Elemento opzionale di `Fault`.

**Il `Value` figlio di `Code` ha una sintassi diversa da quella del figlio di `Subcode`.

RPC con SOAP

SOAP permette in particolare di effettuare RPC.

La codifica di una RPC con SOAP segue alcune regole convenzionali, riguardanti:

- L'indicazione della **risorsa** (oggetto) alla quale è indirizzata la chiamata;
- L'indicazione del **metodo** da invocare;
- L'eventuale trasmissione della **signature** del metodo, per una sua più corretta identificazione.
- La trasmissione dei **parametri** del metodo, e del valore di ritorno dello stesso.

RPC con SOAP

L'indicazione della risorsa (oggetto) alla quale è indirizzata la chiamata avviene in maniera **dipendente dal protocollo** usato per trasportare il messaggio.

- Usando HTTP, sarà la URI richiesta al server a specificare l'oggetto del quale si vuole invocare un determinato metodo.
- Convenzionalmente, l'header field SOAPAction conterrà la URI completa dell'oggetto seguita dal nome del metodo.

RPC con SOAP

- La chiamata al metodo viene codificata come una struttura.
- L'elemento root della struttura dovrà avere lo stesso nome del metodo e un tipo uguale alla *signature* dello stesso.
  - I parametri saranno **codificati come membri della struttura**, elencati nello stesso ordine con cui compaiono nella signature del metodo.
  - Il loro accessor sarà un elemento con **lo stesso nome e tipo del parametro formale** del metodo.

RPC con SOAP - Esempio

```
public class HelloWorld {
  public String SayHelloTo(String name, int hh) {
    if (hh>18)
      return " Buona Sera " + name;
    else
      return " Buon Giorno " + name;}
}
```

Questo è il metodo che intendiamo chiamare via SOAP.

```
<xs:element name="SayHelloTo">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="hh" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="SayHelloToResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="return" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Fase 1. Codifica della signature del metodo.

In questa fase usiamo la codifica standard di SOAP per descrivere lo schema degli elementi che incapsulano la chiamata al metodo e la

risposta associata.

Questo schema verrà associato al namespace http://foo.unige.it/helloworld

RPC con SOAP - Esempio

```
<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:SayHelloTo
xmlns:m="http://foo.unige.it/helloworld">
      <name>Giuseppe</name>
      <hh>16</hh>
    </m:SayHelloTo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fase 2. Creazione del messaggio SOAP.

Il messaggio contenente la RPC viene creato seguendo le regole sintattiche di SOAP. Non è necessario specificare i tipi di <name> e <hh> tramite l'attributo xsi:type, perché lo schema per questi elementi lo specifica in maniera univoca.

```
POST /HelloWorld HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: 321
SOAPAction: "http://foo.unige.it/helloworld#SayHelloTo"

<SOAP-ENV:Envelope
  …
</SOAP-ENV:Envelope>
```

Fase 3. Creazione della request HTTP.

Viene specificato il nome dell'oggetto da chiamare, e viene attaccato un payload contenente il messaggio preparato nella fase precedente.

RPC con SOAP

- L'esito della chiamata, se positivo, viene codificato come una struttura:
  - L'elemento root della struttura ha convenzionalmente il nome del metodo seguito da "Response".
  - Il primo elemento della struttura è il **valore di ritorno del metodo**. Il suo nome non ha importanza.
  - A seguire, vengono inseriti, nell'ordine con cui compaiono nella *signature*, i valori di tutti i **parametri di tipo [out] e [in/out]** del metodo.

RPC con SOAP - Esempio

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 615

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encodin
g/">
  <SOAP-ENV:Body>
    <m:SayHelloToResponse
xmlns:m="http://foo.unige.it/helloworld">
      <return>Buona Sera Giuseppe</return>
    </m:SayHelloTo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

l'elaborazione della richiesta ha avuto successo.

Fase 4. Ricezione della risposta via HTTP. Il server invia la risposta elaborata dal metodo che abbiamo richiesto. La risposta è stata formattata come messaggio SOAP seguendo la codifica data dallo Schema visto nella Fase 1.Il server invia il messaggio SOAP di risposta come fosse un normale contenuto HTTP.

Creare Web Services

Gli esempi che vedremo sono implementati usando:

- **Java**
- **Tomcat** un server Web ottimizzato per eseguire programmi Java lato server (**servlet**), per cui ci si riferisce ad esso indicandolo come un **contenitore di servlet**. Per il download, vedere
  `http://jakarta.apache.org/tomcat/`
  versione consigliata **Tomcat 4.1.x**.
- **Axis** e' a sua volta un servlet particolare che sa come trattare i protocolli SOAP, Xml Schema e WDSL.
  Per l'installazione vedere
  `http://ws.apache.org/axis/java/install.html`

Premesse sull'Installazione 1

- Nell'installazione di Tomcat si trova una directory webapps in cui vanno collocate le applicazioni web. In questa directory va copiata la directory webapps/axis dalla distribuzione xml-axis . Questa directory può chiamarsi in qualunque modo, ma il nome scelto sarà la base dell'URL attraverso il quale i client potranno consultare il servizio. Noi supporremo che tale directory si chiami axis
- Nella directory axis, si trova una subdirectory WEB-INF. Questa directory, oltre a contenere alcune configurazioni di base, verrà usata per contenere i web services che costruiremo e vorremo rendere disponibili.

Premesse sull'Installazione 2

- Axis deve poter trovare un parser XML. Se il Web server o Java runtime non ne rendono uno visibile alle applicazioni web, è necessario scaricarlo e aggiungerlo. Java 1.4 comprende il parser Crimson, perciò si può omettere questo passaggio.
- CATALINA_HOME è il nome della variabile d'ambiente contenente il percorso alla directory dove è stato installato il Web server (TOMCAT). Avvio il webserver da linea di comando con `%CATALINA_HOME%/bin/startup`
- La pagina di inizio delle applicazioni web è `http://127.0.0.1:8080/axis/`

Esempio1: il Servizio

- Creo il file:

```
public class CiaoATutti {
      public String ciao() {
         String results = "Ciao a tutti";
      return results;
    }
}
```

- Lo copio in `%CATALINA_HOME%\webapps\axis` come `CiaoATutti.jws`
- Oss: è già accessibile a http://127.0.0.1:8080/axis/CiaoATutti.jws?method=ciao

http://127.0.0.1:8080/axis/CiaoATutti.jws?method=ciao

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

     <soapenv:Body>
       <ciaoResponse     soapenv:encodingStyle=

   "http://schemas.xmlsoap.org/soap/encoding/">
           <ciaoReturn xsi:type="xsd:string">Ciao a tutti</ciaoReturn>
       </ciaoResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

http://127.0.0.1:8080/axis/CiaoATutti.jws?wsdl

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://127.0.0.1:8080/axis/CiaoATutti.jws" xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://127.0.0.1:8080/axis/CiaoATutti.jws"
      xmlns:intf="http://127.0.0.1:8080/axis/CiaoATutti.jws" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:message name="ciaoRequest" />
- <wsdl:message name="ciaoResponse">
  <wsdl:part name="ciaoReturn" type="xsd:string" />
  </wsdl:message>
- <wsdl:portType name="CiaoATutti">
- <wsdl:operation name="ciao">
  <wsdl:input message="impl:ciaoRequest" name="ciaoRequest" />
  <wsdl:output message="impl:ciaoResponse" name="ciaoResponse" />
  </wsdl:operation>
  </wsdl:portType>
- <wsdl:binding name="CiaoATuttiSoapBinding" type="impl:CiaoATutti">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="ciao">
  <wsdlsoap:operation soapAction="" />
- <wsdl:input name="ciaoRequest">
  <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://DefaultNamespace" use="encoded" />
  </wsdl:input>
- <wsdl:output name="ciaoResponse">
  <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://127.0.0.1:8080/axis/CiaoATutti.jws" use="encoded" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:service name="CiaoATuttiService">
- <wsdl:port binding="impl:CiaoATuttiSoapBinding" name="CiaoATutti">
  <wsdlsoap:address location="http://127.0.0.1:8080/axis/CiaoATutti.jws" />
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

Osservazioni

- Un servizio realizzato con un file .jws richiede per la sua messa in opera solo il file suddetto e viene trasformato automaticamente in servizio Web quando e' copiato in una qualsiasi delle cartelle di axis

  (escluso il sottoalbero WEB-INF).
- Per tutti i servizi e' possibile vedere il WSDL usando l' URL

  `http://127.0.0.1:8080/axis/`NOME-DEL-FILE`.jws?wsdl`

Esempio1: Client

```
import org.apache.axis.client.Service;
   import org.apache.axis.encoding.XMLType;
   import javax.xml.rpc.*;
   public class CiaoClient
   {
     public static void main(String [] args) throws Exception {
     String host = "http://localhost:8080";
     String servicepath = "/axis/CiaoATutti.jws";
     String endpoint = host   + servicepath;
     String method = "ciao" ;

     String ret = null;
     Service service = new Service();
     Call call = (Call) service.createCall();

     call.setTargetEndpointAddress(endpoint);
     call.setOperationName(new javax.xml.namespace.QName(method));

     ret = (String) call.invoke(( Object [] )null);
     System.out.println("Got result : " + ret);
      }
   }
```

Configurare il CLASSPATH per la compilazione

- **UNIX:**

```
setenv CLASSPATH
$JAVA_HOME/lib:.:$AXIS_HOME/lib/axis.jar:$AXIS_
HOME/lib/commons-
logging.jar:$AXIS_HOME/lib/jaxrpc.jar:$AXIS_HOM
E/lib/log4j-
1.2.8.jar:$AXIS_HOME/lib/jaxrpc.jar:$AXIS_HOME/
lib/commons-
discovery.jar:$AXIS_HOME/lib/saaj.jar:$CATALINA
_HOME/common/endorsed/xercesImpl.jar:$CATALINA_
HOME/common/endorsed/xmlParserAPIs.jar:$AXIS_HO
ME/lib/wsdl4j.jar
```

- **WINDOWS:**

  stessi percorsi separati da “ ; ”

  le variabili d’ ambiente vanno indicate tra %

Esempio1: Esecuzione

- Il client è copiato in AXIS_HOME;
- Compilazione del client;
- Esecuzione del client: `java CiaoClient`

```
Got result : Ciao a tutti
```

Esempio2: Servizio e wsdl

```
public class Plus { public int plus(int
  a, int b) { return a+b; } }
```

- Copio come Plus.jws in `%CATALINA_HOME%\webapps\axis`
- Il wsdl è immediatamente disponibile alla pagina `http://127.0.0.1:Plus.jws?wsdl`
- La pagina `http://127.0.0.1:Plus.jws?method=plus` questa volta è vuota.

Esempio2: Client

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import javax.xml.rpc.ParameterMode;
public class SommaClient {
   public static void main(String [] args)
        throws Exception
          {String host = "http://localhost:8080";
           String servicepath = "/axis/Somma.jws";
           String endpoint = host + servicepath;
           String method = "somma" ;
           String ret = null;
           Service service = new Service();
           Call call = (Call) service.createCall();
           call.setTargetEndpointAddress(new java.net.URL (endpoint));
           call.setOperationName(method);
           Integer op1=new Integer(args[0]);
           Integer op2=new Integer(args[1]);
           call.addParameter("op1", XMLType.XSD_INTEGER,
   ParameterMode.IN); call.addParameter("op2", XMLType.XSD_INTEGER,
   ParameterMode.IN);
           ret =((Integer)call.invoke(newObject{op1,op2})).toString();
           System.out.println("Got result : " + ret); }
}
```

File di deployment

```
<deployment
     xmlns="http://xml.apache.org/axis/wsdd/"

   xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
   >

        <service name="Plus" provider="java:RPC">
          <parameter name="className"
   value="esempi.plusservice.Plus"/>
          <parameter name="allowedMethods" value="*"/>
        </service>
</deployment>
```

Copiato in axis come deployment.wsdd

WSDD Web Service Deployment Descriptor

Messa in opera del servizio

```
package esempi.plusservice;
public class Plus { public int plus(int a, int b)
   { return a+b; } }
```

- Salvo come Plus.java;
- Compilo;
- Creo una directory `CATALINA_HOME/webapps/axis/WEB-INF/classes/esempi/`
- Creo una directory `CATALINA_HOME/webapps/axis/WEB-INF/classes/esempi/plusservice` e vi copio Plus.class
- Viene dato il comando

  ```
  java org.apache.axis.client.AdminClient
  deploy.wsdd
  ```

http://127.0.0.1:8080/axis/Plus.jws?method=plus&in0=3&in1=5

```
<?xml version="1.0" encoding="UTF-8" ?>
-<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
-<soapenv:Body>
-<plusResponse
   soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <plusReturn xsi:type="xsd:int">8</plusReturn>
   </plusResponse>
   </soapenv:Body>
   </soapenv:Envelope>
```

Riferimenti

Simon St. Laurent, Joe Johnstone e Edd Dumbill
Programmare Web Services con XML-RPC
Hops libri- collana O'Reilly

Specifica della versione 1.2 di SOAP-parte sul Messaging
http://www.w3.org/TR/2003/PR-soap12-part1-20030507/

Specifica di SOAP 1.1 dal W3C
http://www.w3.org/TR/SOAP/

Tutorial su SOAP + Java su JavaWorld
http://www.javaworld.com/javaworld/jw-03-2001/jw-0330-soap_p.html